Countering Unauthorized Code Execution on Commodity Kernels: A Survey of Common Interfaces Allowing Kernel Code Modification[☆]

Trent Jaeger

Department of Computer Science and Engineering, 346A IST Building, University Park, PA 16802

Paul C. van Oorschot

1125 Colonel By Drive, Ottawa, Ontario K1S 5B6 Canada

Glenn Wurster

1125 Colonel By Drive, Ottawa, Ontario K1S 5B6 Canada

Abstract

Motivated by the goal of hardening operating system kernels against rootkits and related malware, we survey 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 survey 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, Survey

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 (ring 0) processor control. Typically, whether code is running with supervisor level processor control is indicated by one or more bits in the processor which denote the privileges given to running code - all user-space applications will run with less privileges than the kernel. 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 assume the attacker has super-user access on a system (i.e., they can run arbitrary code as the most privileged

[☆]Version: August 10, 2011. Authors in alphabetical order. Contact author: glenn@wurster.ca

Email addresses: tjaeger@cse.psu.edu (Trent Jaeger), paulv@scs.carleton.ca (Paul C. van Oorschot), glenn@wurster.ca (Glenn Wurster)

user on a system – typically root on a Linux system) and is attempting to run arbitrary code with supervisor processor control. We also assume that the attacker is remote – physical attacks against the system [8], including direct memory, IO, or storage access, and the addition of new hardware (e.g., hooking into the memory bus or moving the storage media to another system controlled by the attacker) are beyond the scope of this survey. The threat model we use herein is the common one used by most user-space DAC and MAC application privilege schemes implemented in the OS kernel – a user-space application should not be able to bypass the access control scheme by modifying the enforcement mechanism in the kernel.

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 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 [55, 56, 77, 47, 17, 31, 78, 67, 26, 11]. 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., [27, 14]), or "live CD" (boot CD) approaches which run new external bootable operating systems by changing the boot order and booted 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. The processor will typically provide mechanisms which prevent user-space applications from modifying the kernel running with supervisor level processor control. To modify the kernel, therefore, a user-space application must request the modification by communicating with the kernel (illustrated by the arrows crossing the protection boundary in Figure 1). As discussed above, we assume an attacker has super-user access on a system and is attempting to get supervisor level processor control. Once malware has obtained this elevated privilege, there are many opportunities [35]. Unlike other studies of kernel malware, we focus our survey on how malware manages to obtain access, not what the malware can do once access has been obtained.

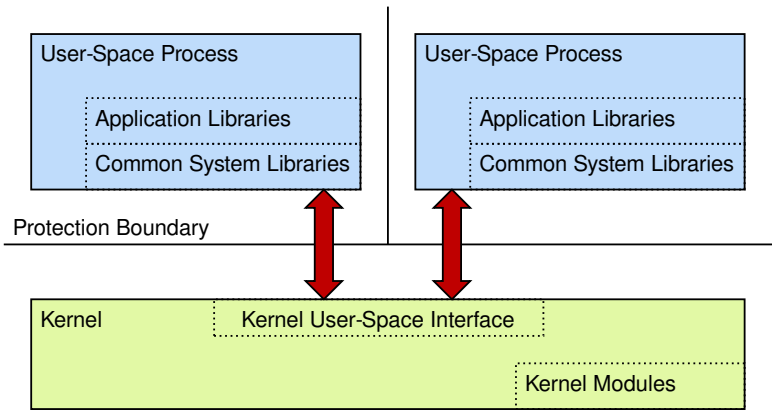

Figure 1: Protection Boundary between the kernel (which runs with supervisor level processor control), and user-space processes

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 [60, 48, 34]. 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 [71]. 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 [61] 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 [18].

2.3. Memory Access Interfaces

The kernel may export to applications an interface allowing arbitrary access to the physical address space of the machine [76, 75, 60, 63]. 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) [59]. 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` [21].

[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/

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 [14, 27].

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) [10, 33], FireWire [28], and ACPI [32] may provide mechanisms for updating physical memory. 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 2, where ultimately the request to modify the kernel travels through the kernel on the way to the hardware.

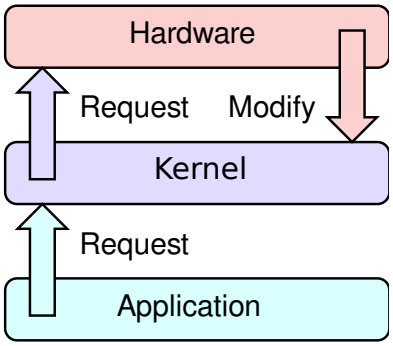

Figure 2: Request flow for modifying the kernel via hardware

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 [79].

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 [50] 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., CPU and display power management [73]).

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 which will be run during the boot process, the application can modify the kernel which is run after the next reboot. Such an approach would typically resemble the following steps: 1) Modify the kernel or boot loader on disk; 2) Cause the system to reboot; 3) When the system comes back after the reboot, the modified kernel will be running. 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

[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.

alter the kernel [33]). Depending on the firmware used during boot, the boot loader responsible for loading and executing the kernel may exist in EEPROM [68], in the boot sector [5], or in a partition [29, 81].

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 [1] 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 [57, 52]. 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) [22, 23].

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 [35, 45], e.g., hooking system calls [38], hiding processes [78], spawning a virtual machine monitor (e.g., SubVirt [44] and Bluepill [61]) and extracting authentication credentials [42].

Our survey is concerned with those interfaces which are intentionally made available to user-space processes by the kernel. We do not concern ourselves with attack vectors relying on kernel vulnerabilities. Many protection mechanisms, including ASLR and stack cookies (see related work discussion in other recent papers [43, 37]), can be made to protect kernels as well as applications [24]. Dynamic and static analysis [12, 15] tools can also be used to locate kernel vulnerabilities.

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 [46, 60]. While not currently accepted into the mainline Linux kernel, a patch does exist to implement this feature [46]. 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 [60, 46]. 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, the integrity of kernel memory can be protected. This includes restricting arbitrary writes to both the swap file, as well as to those sectors of the disk occupied by swap (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 [51]. 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 [60].

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 [75]. 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 [76].

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 [21]. Any attempt to access this device node by a user-space application, however, has been denied since Windows 2003 SP1 [60, 49].

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) [7, 6] 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 [4]. Approach 1 has also been attempted with read-only media (e.g., booting off a CD [65]). 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 [36], and kernel module signing has been used (see Section 3.1). We discuss approach 2 further in Section 4.1. To prevent booting from a storage device not protected by the kernel (e.g., a CD drive), the boot device priority can be configured in BIOS (or EFI) so that it is hard-coded which device will be attempted first for boot. By setting BIOS to use the device containing the legitimate kernel, that kernel will always be booted. The BIOS setting itself is protected by either requiring a reboot to change it (changing the BIOS, which requires the attacker be physically present, is outside our threat model), or hardware access (which is moderated by the currently running kernel – see Section 2.4).

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`. `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 [30, 25] and address space layout randomization [66, 20]). 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 [58]. While super-user processes can disable this mechanism, most user-space processes are now prevented from exploiting a null pointer dereference 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 Secure Boot

Secure boot approaches can be used to verify the boot loader and operating system as it is loaded [9, 53]. Secure boot involves having each component in the boot process compare a cryptographic hash of the next component to-be-run to a known-good value. If the computed hash is equal to a known-good value, the next component is executed, otherwise the currently running component avoids running the component for which verification failed (this is in contrast to authenticated boot, where each component is hashed, but execution of a component is not prevented). Using secure boot, 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 or modified [9]). Without disabling other methods of modifying the kernel, however, a verified 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 [62], or all other methods of modifying the kernel, as discussed in Sections 3.1 through 3.7 above, must be restricted. The trusted computing platform [3, 2], which can be used to create a measurement chain [41, 74], attempts to detect rather than prevent modification of the kernel – allowing authenticated boot.

4.2. Using Fine-Grained Mandatory Access Control

AppArmor [13, 19] and SELinux [40, 69] 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 [39], 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 [64], 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. [16] 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. [54] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. Strunk et al. [72] proposed logging all file modifications for a period of time to assist in the recovery after malware infection.

4.4. Using VMMs or Other Hardware

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 [55], 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. [56] likewise use a system device to detect changes in the OS kernel, but concentrate on protecting dynamic data structures.

Carbone et al. [17] 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 (VMM) technology to either detect or prevent kernel malware (e.g., see recent papers [70, 80]).

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

References

[1] ECMA 107: Volume and File Structure of Disk Cartridges for Information Interchange. ECMA; 2nd ed.; 1995. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-107.pdf.
[2] Trusted Computing Platforms: TCPA Technology in Context. Prentice Hall, 2002.
[3] TCG Specification Architecture Overview. Trusted Computing Group, Inc.; version 1.4 ed.; 2003. .
[4] Built-In Anti-Virus Support in Windows 95. Microsoft; 1st ed.; 2006. http://support.microsoft.com/kb/q143281/.
[5] Modern Operating Systems; Prentice Hall. 3rd ed.
[6] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., Wiegert, J.. Intel virtualization technology for directed I/O. Intel Technology Journal 2006;http://www.intel.com/technology/itj/2006/v10i3/.
[7] AMD. AMD I/O Virtualization Technology (IOMMU) Specification. Advanced Micro Devices, Inc.; revision 1.26 ed.; 2009. .

[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.

[8] Anderson, R., Kuhn, M.. Tamper resistance - a cautionary note. In: Proc. 2nd USENIX Workshop on Electronic Commerce. 1996. .
[9] Arbaugh, W.A., Farber, D.J., Smith, J.M.. A secure and reliable bootstrap architecture. In: Proc. 18th IEEE Symposium on Security and Privacy. 1997. p. 65–71.
[10] Baer, J.L.. Computer Systems Architecture. Computer Science Press, 1980.
[11] Baliga, A., Chen, X., Iftode, L.. Paladin: Automated Detection and Containment of Rootkit Attacks. Technical Report DCS-TR-593; Rutgers Univ. Dpt. of Computer Science; 2006.
[12] Ball, T.. The concept of dynamic analysis. In: Proc. 7th European Software Engineering Conference. volume 26; 1999. .
[13] Bauer, M.. Paranoid penguin: an introduction to Novell AppArmor. In: Linux Journal. Number 148; 2006. p. 13.
[14] Boileau, A.. Hit by a bus: Physical access attacks with firewire. Presentation, Ruxconn; 2006.
[15] Bush, W.R., Pincus, J.D., Sielaff, D.J.. A static analyzer for finding dynamic programming errors. Software - Practice and Experiance 2000;(30):775–802.
[16] Butler, K.R.B., McLaughlin, S., McDaniel, P.D.. Rootkit-resistant disks. In: Proc. 15th ACM Conference on Computer and Communications Security. 2008. p. 403–415.
[17] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.. Mapping kernel objects to enable systemic integrity checking. In: Proc. 16th ACM Conference on Computer and Communications Security. 2009. .
[18] Chanet, D., Cabezas, J., Morancho, E., Navarro, N., Bosschere, K.D.. Linux kernel compaction through cold code swapping. In: Transactions on High-Performance Embedded Architectures and Compilers II. 2009. p. 173–200.
[19] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V.. Subdomain: Parsimonious server security. In: Proc. LISA '00: 14th Systems Administration Conference. 2000. p. 341–354.
[20] Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: DARPA Information Survivability Conference and Expo. 2000. p. 119–129.
[21] crazylord, . Playing with Windows /dev/(k)mem. In: Phrack. volume 0x0b (0x3b); 2002. http://www.phrack.com/issues.html?issue=59&id=16.
[22] cve-2008-0010. Vulnerability summary for cve-2008-0010 - copy_from_user_mmap_sem. CVE; 2008. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0010.
[23] cve-2010-3081. Vulnerability summary for cve-2010-3081 - compact_alloc_user_space. CVE; 2010. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3081.
[24] Dalton, M., Kannan, H., Kozyrakis, C.. Real-world buffer overflow protection for userspace & kernelspace. In: Proc. 17th USENIX Security Symposium. 2008. p. 395–410.
[25] Das, M., Lerner, S., Seigle, M.. ESP: path-sensitive program verification in polynomial time. In: Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. 2002. p. 57–68.
[26] Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J.. Robust signatures for kernel data structures. In: Proc. 16th ACM Conference on Computer and Communications Security. 2009. .
[27] Dornseif, M.. Owned by an ipod. Presentation, PacSec; 2004.
[28] Dornseif, M.. Owned by an iPod. In: PacSec. 2004. .
[29] EFI. Unified Extensible Firmware Interface Specification. Unified EFI, Inc.; version 2.3, errata b ed.; 2010. .
[30] Evans, D., Larochelle, D.. Improving security using extensible lightweight static analysis. In: IEEE Software. IEEE Computer Society; number 1 in 19; 2002. p. 42–51.
[31] Garfinkel, T., Rosenblum, M.. A virtual machine introspection based architecture for intrusion detection. In: Proc. 2003 Network and Distributed Systems Security Symposium. Internet Society; 2003. p. 191–206.
[32] Heasman, J.. Implementing and detecting an ACPI BIOS rootkit. In: Proc. Blackhat Federal. 2006. .
[33] Heasman, J.. Implementing and detecting a PCI rootkit. In: Proc. Blackhat Federal. 2007. .
[34] Henderson, B.. Linux Loadable Kernel Module HowTo; v1.06 ed.; 2005. .
[35] Hoglund, G., Butler, J.. Rootkits: Subverting the Windows Kernel. Addison-Wesley, 2005.
[36] Huang, A.. Hacking the Xbox. No Starch Press, Inc., 2003.
[37] Hund, R., Holz, T., Freiling, F.C.. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proc. 18th USENIX Security Symposium. 2009. p. 383–398.
[38] Hunt, G., Brubacher, D.. Detours: Binary interception of Win32 functions. In: Proc. 3rd USENIX Windows NT Symposium. 1999. .
[39] IEEE. 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.
[40] Jaeger, T., Sailer, R., Zhang, X.. Analyzing integrity protection in the SELinux example policy. In: Proc. 12th USENIX Security Symposium. 2003. p. 59–74.
[41] Kauer, B.. OSLO: Improving the security of trusted computing. In: Proc. 16th USENIX Security Symposium. 2007. p. 229–237.
[42] keylogger. How to: Building your own kernel space keylogger. Web Page; 2010. http://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html.
[43] Kil, C., Jun, J., Bookholt, C., Xu, J.. Address space layout permutation. In: Proc. 22nd Applied Computer Security Applications Conference. 2006. p. 339–348.
[44] King, S., Chen, P., Wang, Y.M., Verbowski, C., Wang, H., Lorch, J.. Subvirt: Implementing malware with virtual machines. In: Proc. 2006 IEEE Symposium on Security and Privacy. 2006. p. 314–327.
[45] Kong, J.. Designing BSD Rootkits: An Introduction to Kernel Hacking. No Starch Press, 2007.
[46] Kroah-Hartman, G.. Signed kernel modules. Linux Journal 2004;117:48–53.
[47] Kruegel, C., Robertson, W., Vigna, G.. Detecting kernel-level rootkits through binary analysis. In: Proc. 20th Annual Computer Security Applications Conference (ACSAC'04). IEEE Computer Society; 2004. p. 91–100.
[48] Love, R.. Linux Kernel Development. 2nd ed. Novell Press, 2005.

[49] Microsoft Corporation, . Device\PhysicalMemory object. TechNet Article (viewed 20 Feb 2010). http://technet.microsoft.com/en-us/library/cc787565(WS.10).aspx.
[50] Mochel, P.. The sysfs filesystem. In: Proc. Ottawa Linux Symposium. 2005. .
[51] MSDN. WriteFileEx function. Web Page; 2008. http://msdn.microsoft.com/en-us/library/aa365748(VS.85).aspx.
[52] Nellitheertha, H.. Reboot Linux faster using kexec. IBM; 2004. http://www.ibm.com/developerworks/linux/library/l-kexec/index.html.
[53] Parno, B., McCune, J.M., Perrig, A.. Bootstrapping trust in commodity computers. In: Proc. 2010 IEEE Symposium on Security and Privacy. 2010. p. 414–429.
[54] Pennington, A., Strunk, J., Griffin, J., Soules, C., Goodson, G., Ganger, G.. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In: Proc. 12th USENIX Security Symposium. 2003. p. 137–151.
[55] Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.. Copilot - a coprocessor-based kernel runtime integrity monitor. In: Proc. 13th USENIX Security Symposium. 2004. p. 179–194.
[56] Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proc. 15th USENIX Security Symposium. 2006. p. 289–304.
[57] Pfiffer, A.. Reducing System Reboot Time With kexec. Open Source Development Labs, Inc.; 2003. .
[58] Red Hat, Inc., . How to mitigate against null pointer dereference vulnerabilities? Web Page; 2010. https://access.redhat.com/kb/docs/DOC-20536.
[59] Reilly, E.D.. Encyclopedia of Computer Science; John Wiley and Sons, Ltd. 4th ed.; p. 1152.
[60] Ruff, N.. Windows memory forensics. Journal in Computer Virology 2008;4(2):83–100.
[61] Rutkowska, J.. Subverting Vista kernel for fun and profit. Blackhat Presentation; 2006. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.
[62] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.. Design and implementation of a TCG-based integrity measurement architecture. In: Proc. 13th USENIX Security Symposium. 2004. p. 223–238.
[63] sd, , devik, . Linux on-the-fly kernel patching without LKM. In: Phrack. volume 0x0b (0x3a); 2001. http://www.phrack.org/issues.html?issue=58&id=7.
[64] SELinux. SELinux reference policy. Web Page (accessed 31 Dec 2010). http://oss.tresys.com/projects/refpolicy.
[65] Server2Go. Server2go. Web Page; 2011. http://www.server2go-web.de/.
[66] Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.. On the effectiveness of address-space randomization. In: Proc. 11th ACM Conference on Computer and Communications Security. 2004. p. 298–307.
[67] Sharif, M., Lee, W., Cui, W.. Secure in-VM monitoring using hardware virtualization. In: Proc. 16th ACM Conference on Computer and Communications Security. 2009. .
[68] Siemsen, R.. The NetWinder Firmware HOWTO; 2001. http://www.netwinder.org/howto/Firmware-HOWTO-all.html.
[69] Smalley, S., Vance, C., Salamon, W.. Implementing SELinux as a Linux Security Module. Technical Report 01-043; NAI Labs; 2002.
[70] Srivastava, A., Giffin, J.. Efficient monitoring of untrusted kernel-mode execution. In: Proc. 2011 Network and Distributed Systems Security Symposium. Internet Society; 2011. .
[71] Stallings, W.. Operating Systems: Internals and Design Principles. 4th ed. Prentice Hall, 2001.
[72] Strunk, J., Goodson, G., Scheinholtz, M., Soules, C., Ganger, G.. Self-securing storage: Protecting data in compromised systems. In: Proc. 4th Symposium on Operating Systems Design and Implementation. 2000. .
[73] sysfs. Processor - scheduler tunables for multi-socket systems. Web site (viewed 03 Jul 2011); 2004. http://www.lesswatts.org/tips/cpu.php.
[74] tboot. Trusted boot open source project. Web site (viewed 13 Mar 2011). http://sourceforge.net/projects/tboot/.
[75] van de Ven, A.. make /dev/kmem a config option. GIT Commit; 2008. http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=b781ecb6a379f155568ef7093e38c6c1d857fe53.
[76] van de Ven, A.. x86: Introduce /dev/mem restrictions with a config option. GIT Commit; 2008. http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=ae531c26c5c2a28ca1b35a75b39b3b256850f2c8.
[77] Wang, Y.M., Beck, D., Vo, B., Roussev, R., Verbowski, C.. Detecting stealth software with strider ghostbuster. In: Proc. International Conference on Dependable Systems and Networks (DSN-DCCS). 2005. p. 368–377.
[78] Wang, Z., Jiang, X., Cui, W., Ning, P.. Countering kernel rootkits with lightweight hook protection. In: Proc. 16th ACM Conference on Computer and Communications Security. 2009. .
[79] Wojtczuk, R., Ruthowska, J.. Attacking SMM Memory via Intel CPU Cache Poisoning. Invisible Things Lab; 2009. http://www.invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.
[80] Xiong, X., Tian, D., Liu, P.. Practical protection of kernel integrity for commodity OS from untrusted extensions. In: Proc. 2011 Network and Distributed Systems Security Symposium. Internet Society; 2011. .
[81] Zimmer, V., Rothman, M., Hale, R.. Beyond BIOS: Implementing the Unified Extensible Firmware Interface with Intel's Framework. Intel Press, 2006.