Rootkit-Resistant Disks

Kevin R. B. Butler, Stephen McLaughlin and Patrick D. McDaniel
Systems and Internet Infrastructure Security Laboratory (SIIS)
Pennsylvania State University, University Park, PA
{butler,smclaugh,mcdaniel}@cse.psu.edu

ABSTRACT

Rootkits are now prevalent in the wild. Users affected by rootkits are subject to the abuse of their data and resources, often unknowingly. Such malware becomes even more dangerous when it is *persistent*–infected disk images allow the malware to exist across reboots and prevent patches or system repairs from being successfully applied. In this paper, we introduce rootkit-resistant disks (RRD) that label all immutable system binaries and configuration files at installation time. During normal operation, the disk controller inspects all write operations received from the host operating system and denies those made for labeled blocks. To upgrade, the host is booted into a safe state and system blocks can only be modified if a security token is *attached to the disk controller*. By enforcing immutability at the disk controller, we prevent a compromised operating system from infecting its on-disk image.

We implement the RRD on a Linksys NSLU2 network storage device by extending the I/O processing on the embedded disk controller running the SlugOS Linux distribution. Our performance evaluation shows that the RRD exhibits an overhead of less than 1% for filesystem creation and less than 1.5% during I/O intensive Postmark benchmarking. We further demonstrate the viability of our approach by preventing a rootkit collected from the wild from infecting the OS image. In this way, we show that RRDs not only prevent rootkit persistence, but do so in an efficient way.

Categories and Subject Descriptors

D.4.6 **[Operating Systems]**: Security and Protection

General Terms

Security

Keywords

storage, security, rootkits, labels

1. INTRODUCTION

Rootkits exploit operating system vulnerabilities to gain control of a victim host. For example, some rootkits replace the system call table with pointers to malicious code. The damage is compounded when such measures are made *persistent* by modifying the on-disk system image, e.g., system binaries and configuration. Thus, the only feasible way of recovering from a rootkit is to wipe the disk contents and reinstall the operating system [3, 13, 19, 20]. Worse still, once installed, it is in almost all cases impossible to securely remove them. The availability of malware and the economic incentives for controlling hosts has made the generation and distribution of rootkits a widespread and profitable activity [44].

Rootkit-resistant operating systems do not exist today, nor are they likely to be available any time soon; to address rootkits is to largely solve the general problem of malicious software. Current operating system technologies provide better tools than previously available at measuring and governing software [34], but none can make the system impervious to rootkits without placing unreasonable restrictions on their operation. However, while it is currently infeasible to prevent an arbitrary rootkit from exploiting a given system, we observe that preventing them from being becoming persistent is a significant step in limiting both their spread and damage.

We introduce a *rootkit-resistant disk* (RRD) that prevents rootkit persistence. We build on increasingly available intelligent disk capabilities to tightly govern write access to the system image within the embedded disk processor. Because the security policy is enforced at the disk processor (rather than in the host OS), a successful penetration of the operating system provides no access to modify the system image. The RRD works as follows:

1. An administrative token containing a *system write capability* in placed in the USB port of the external hard drive enclosure during the installation of the operating system. This ensures that the disk processor has access to the capability, but the host CPU does not.
2. Associated with every block is a label indicating whether it is immutable. Disk blocks associated with immutable system binaries and data are marked during system installation. The token is removed at the completion of the installation.
3. Any modification of an immutable system block during normal operation of the host OS is blocked by the disk processor.
4. System upgrades are performed by safely booting the system with the token placed in the device (and the *system write capability* read), and the appropriate blocks marked. The token is removed at the completion of the upgrade.

An RRD superficially provides a service similar to that of "live-OS" distributions, i.e., images that boot off read-only devices such as a CD. However, an RRD is a significant improvement over such approaches in that (*a*) it can intermix and mutable data with immutable data, (*b*) it avoids the often high overheads of many read-

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

only devices, and (*c*) it permits (essential) upgrading and patching. In short, it allows the host to gain the advantages of a tamper-resistant system image without incurring the overheads or constraints of read-only boot media.

In this paper, we present the design and analysis of the RRD. The system architecture, implementation, and evaluation are detailed and design alternatives that enable performance and security optimizations discussed. We implement the RRD on a Linksys NSLU2 network storage device [33] by extending the I/O processing on the embedded disk controller, and use USB flash memory devices for security tokens. Our implementation integrates label and capability management within the embedded software stack (SlugOS Linux distribution [50]). We further extend the host operating system kernel and installation programs to enable the use of the non-standard RRD interfaces and security tokens: however, in practice, modifications to host operating systems will not be needed.

Our performance evaluation shows that the RRD exhibits small performance and resource overheads. The experiments show an overhead of less than 1% for filesystem creation and less than 1.5% during I/O intensive Postmark benchmarking. Further investigation shows that the approach imposes a storage overhead of less than 1% of the disk in a worst-case experiment. We experimentally demonstrate the viability of the RRD as a rootkit countermeasure by infecting and recovering from a rootkit collected from the wild. Furthermore, we show through examination of the *chkrootkit* utility that a large number of rootkits would be rendered non-persistent through use of the RRD.

Mutable configuration and binaries that can compromise the system (such as user `cron` [60] jobs), can reinfect the system after reboot. However, once patched, the system will be no longer be subject to the whims of that malware. This represents a large step forward in that it introduces a previously unavailable feasible path toward recovery. Note that the RRD does not protect the system's BIOS (which is burned into system PROM/EPROM/flash). The RRD does, however, protect all portions of the boot process that use immutable code or data, including the master boot record (MBR).

Section 2 introduces in more detail the concepts behind RRDs and their design. Section 3 describes the implementation of an RRD, while Section 4 describes our performance evaluation, Section 5 discusses practical issues when using RRDs, and Section 6 provides related work. We conclude with Section 7.

2. ROOTKIT-RESISTANT DISKS

To understand the requirements for designing storage solutions that resist persistent rootkits, we first examine their nature and operation and commonalities that exist between them. We first present a background on how rootkits have operated to date, then layout requirements for disks that prevent persistent rootkits and the design decisions that we made to implement these goals.

2.1 Background

Rootkits have been well-studied, and those that attack the operating system and reside in the kernel have been demonstrated in both theory and practice [21]. They can be user-mode programs that perform functions such as adding inline hooks into system functions or patching runtime executables dynamically (e.g., system commands such as `ps`, `netstat`, and `top`), or kernel-mode programs that hook into the kernel, layer themselves onto device drivers, or directly manipulate the OS kernel, and sometimes the hardware itself [24]. Rootkits can be persistent, where they survive a system reboot, or non-persistent, where they install themselves into volatile memory and do not survive across reboots [22].

Numerous techniques for hiding rootkits have been implemented, including modification of system files and libraries [8], boot sector modification [51], and altering the ACPI code often stored in the BIOS [23] – this approach may potentially even evade detection by a TPM, by causing it to report correct hash values [26]. While many of these attacks can be fended off through integrity protection mechanisms [27, 40] and kernel-level rootkit detectors [6, 47], increasingly sophisticated rootkits can evade this level of detection. Such attacks can subvert virtual memory [54] or install themselves as a virtual machine monitor underneath the operating system itself [28], demonstrating that whoever controls the lowest layer of the system gains the advantage in attacking or defending it.

With all of these rootkits, a successful compromise means that data is susceptible to exposure. By using RRDs, however, the user can effectively reside at a lower level than the OS by directly interfacing the disk with a physical token to arbitrate access to data. The rootkit will thus be unable to gain access to read and write data on portions of the drive that the user does not have access to, regardless of OS compromise. This provides a level of on-disk protection that has not previously been feasible.

2.2 Goals for an RRD

To provide a practical solution for an RRD, we need to ensure that the following four goals are satisfied:

1. **It must protect against real rootkits.** The RRD must demonstrably protect against currently deployed persistent kernel-level rootkits.

2. **It must be usable without user interaction and with minimal administration.** The operation of the RRD should be *transparent* during normal operation.

3. **It must be highly performant.** Accessing storage must be feasible with as little performance overhead as possible, given the rigorous demands for I/O throughput.

4. **It must have low storage overhead.** The RRD should consume as little ancillary storage for metadata and use as little additional space on the disk as possible.

2.3 RRD Design

Designing a suitable solution that fulfills the above requirements presents the following two challenges:

1. As storage requests travel from a system call to the file system to the storage, context about what is being asked for is lost. For example, knowing whether requests for blocks are related to each other (e.g., are write requests associated with the same file or application) is not possible at the storage layer because this information has been removed. This results in a *semantic gap* between file and storage systems (as described by many, including Sivathanu et al. [49]). Data security policies are often defined at the file level, but the semantic gap makes the task of extending these policies to the disk interface difficult, if not impossible, to implement within conventional operating systems.

2. Enforcement of security in storage independently of the operating system depends on the availability of a trusted administrative interface. The disk interface has traditionally been limited to that of the system bus, as accessible by CPU and possibly DMA controller. This interface is fully accessible to the OS and thus is effectively compromised if the OS is compromised.

We fundamentally address the semantic loss by not relying on the file layer to provide context to the disk. Instead, the *administrator* inserts a token into the disk when data is to be write-protected. The token acts to label the blocks written to disk, such that without the token present, they cannot be overwritten. By doing this, the administrator *provides context to the disk:* it can differentiate between labeled and unlabeled blocks, and between blocks labeled with different tokens. The token may be physically plugged into the drive (e.g., using a smart card or USB token).[1] We say that any data blocks written under a specific token are *bound* to that token, such that they are rendered read-only whenever the token is not present. Such data will be *immutable* to alteration on disk by any processes within the operating system. Only a small subset of the data on a disk will be bound in this manner, notably the binaries and important sectors on a disk (e.g., the MBR) that would otherwise be susceptible to being overwritten by a rootkit. The write-time semantics associated with tokens are a natural consequence, given that administrative operations requiring the presence of tokens are performed on system data at well-defined times (e.g., during file system creation, system installation, and package installation).

The physical action of inserting a physical token addresses our second challenge, as the user is a trusted interface to the disk that cannot be subverted by a compromised operating system. In essence, we have reduced the trust problem to that of physical security of the user and her associated tokens. As previously noted, our model seeks to protect against persistent rootkits that have compromised the operating system; thus, we consider the user a trusted component in the system rather than an adversary. In addition, physical attacks such as forging of the tokens, or attacking the drive itself by opening it to scan its memory for metadata, are outside our protection model. Implementing tamperproof interfaces into the drive appear contrary to the marketplace desires for inexpensive, high-performance storage. However, building additional security into the drive enclosure in a similar manner to the IBM 4758 secure co-processor [12] is a design point that is only feasible to achieve if the cost-benefit ratio for a specific application dictates it to be appropriate.

2.4 Tokens and Disk Policy

An RRD has two modes of operation. Under *normal* operation, the RRD is used like a regular disk, without any tokens present. This is the mode of operation that a regular user will always use the disk in, as will the administrator for the majority of the time. Only during an *administrative event* will the disk be used in *administrator mode*. We define an administrative event to be one that affects the system as a whole and that only an administrator can execute. Examples of these would be the initial installation of an operating system onto the disk and subsequent upgrades to the operating system, e.g., software package upgrades or full distribution upgrades. Administrative mode is activated by inserting a token into the disk. As shown in Figure 1, data blocks written then become labeled with the inserted token and become immutable. Blocks labeled as immutable may only be rewritten when the token associated with the label is inserted into the disk. If the block has not been written under a token, or if it is written without the presence of a token, it is *mutable* and hence not write-protected. By differentiating between mutable and immutable blocks, we can allow certain files such as startup scripts to be only writable in the presence of a token, while not forcing such a stipulation on files that should be allowed to be written, such as log files.

[1]Note that proximity-based solutions such as ZIA [10] do not convey intention and hence would not be suitable for this design.

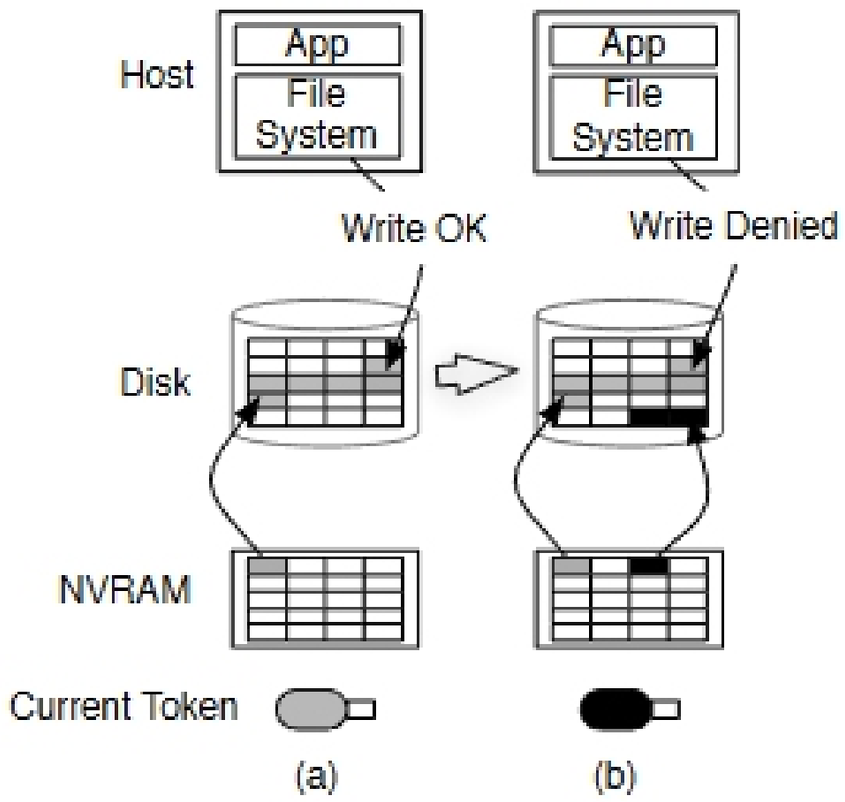

Figure 1: The use of tokens for labeling data in the RRD. Part (a): The host system writes a file to unused disk space using the gray token. The write is allowed and the file is labeled accordingly. In part (b), any blocks written while the black token is in the disk are labeled accordingly, and any attempts to write gray labeled data are denied as long as the gray token is not present.

There is a special token in our system that acts in a manner different than described above. Because of the need for processes to be able to write certain filesystem metadata, such as logs and journals, we introduce the concept of a *permanently mutable* data block. Blocks labeled permanently mutable (denoted ℓ_{pm}) during filesystem creation are writable by all processes (subject to operating system protections, e.g., UNIX permissions), regardless of whether the token is installed or not.

In a scenario where the drive is used to separate binaries that may end up as vectors for rootkits and being loaded at boot time, only one token may be necessary. This token would be used only when system binaries are installed, as that would be the only time they would require being written to the disk. Greater isolation may be achieved by using separate tokens while performing different roles, e.g., a token for installing binaries and another for modifying configuration files. By differentiating between mutable and immutable blocks, we can allow certain files such as startup scripts to be only writable in the presence of a token, while not forcing such a stipulation on files that should be changing, such as document files within a user's home directory.

When the RRD receives a write request for some contiguous region of blocks, $R = \{b_i, b_{i+1}, \ldots b_j\}$, it obtains the label ℓ_t from the current token. If no token is present in the disk then $\ell_t = nil$, in which case the RRD verifies that no mutable blocks are included in R. If ℓ_t is the permanently mutable label ℓ_{pm}, any unlabeled blocks in R are labeled with ℓ_{pm} and the write is allowed. If the token contains any other label, all blocks in the request are checked for equality with that label, and any nil blocks are labeled accordingly. The RRD's write policy is specified in Algorithm 1.

Once a block has been labeled as immutable or permanently mutable, its label cannot be changed. Thus, in a system where immutable data is often written, the possibility of *label creep* [48] arises. Because of the semantic gap between the file and storage layers, we are unable to perform free space reclamation in storage