A Control Point for Reducing Root Abuse of File-System Privileges*

Glenn Wurster
School of Computer Science
Carleton University, Canada
gwurster@scs.carleton.ca

Paul C. van Oorschot
School of Computer Science
Carleton University, Canada
paulv@scs.carleton.ca

ABSTRACT

We address the problem of restricting root's ability to change arbitrary files on disk, in order to prevent abuse on most current desktop operating systems. The approach first involves recognizing and separating out the ability to configure a system from the ability to use the system to perform tasks. The permission to modify configuration of the system is then further subdivided in order to restrict applications from modifying the file-system objects of other applications. We explore the division of root's current ability to change arbitrary files on disk and discuss a prototype that proves out the viability of the approach. Our architecture exposes a control point available for use to enforce policies that prevent one application from modifying another's file-system objects.

1. INTRODUCTION AND OVERVIEW

Major commodity operating systems have a root or superuser account which has total control over the machine, including the file-system. Processes running under the root account can perform arbitrary actions on the file-system, including creating, modifying, and deleting any file-system object (including files, directories, and links). This open environment has lead to many problems, both stability and malware related. On most current desktops, malware uses the ability to change arbitrary files on disk in order to hide itself.

While the principle of least privilege dictates that the privileges assigned to a process should be no more than the privileges required to perform the designed task, the standard exercising of root privilege in order to install applications does not follow the principle. While some progress has been made by encouraging users and daemons not to run as root, the same cannot be said for installers – perhaps the most common use of root privilege in the current computing environment is for system reconfiguration (i.e., installing, uninstalling, or upgrading software). In this paper, we pursue reducing the file-system privileges of root in order to better protect a system against abuse.

The actions performed by any user (including root) on a system can be partitioned into two classes. The first involves actions related to performing day-to-day operations on the system (e.g., writing a paper, browsing, reading email, or playing a game). Such actions typically do not have a lasting impact on the state of the system (modulo data file creation and deletion). The second class involves actions related to changing system configuration. We define the *configuration state* of a system as the set of programs installed, as well as the global configuration related to each program. In order to survive reboot, both the programs installed and all global configuration state must be saved into the file-system, and hence we focus on those configuration operations having a direct visible effect on disk.

The common protection long used in practice is to limit write access to application file-system objects (e.g., files including binaries, directories, symbolic links, and other objects that are part of the file-system) to root [14]. This protection mechanism fails to prevent abuse by applications during install, upgrade, or uninstall. In today's computing environments, it is only realistic to treat any two applications on a system as mutually untrustworthy. Given this updated threat model, we further subdivide configuration in order to *encapsulate* applications – by this we mean that while it may be possible for one application to read the binary, data, and configuration files belonging to another application, it is not possible to modify another's files on disk. In contrast, current desktop approaches for software installation do not prevent an application from modifying or deleting file-system objects related to or created by an unrelated application, a problem previously identified in the literature [53, 48]. Our restriction and division of root file-system permissions addresses this problem, without requiring any radical change in file-system layout (e.g., applications can still install their binaries in a common location such as `/bin`). As a direct result, applications are better protected from malware and other applications, even those running with root privileges.

In our design, which expands on preliminary ideas outlined in a workshop paper [62], the ability to modify arbitrary objects (beyond simply files) on the file-system is removed from root and reassigned to a process running with a new *configuration privilege*. This process in turn can be used to prevent one application from modifying the file-system objects related to another. In creating a distinct configuration permission, the configuration tasks currently performed under root privilege are separated from the everyday tasks. Daemons, applications, or installers running as root no longer automatically inherit configuration privilege.

Our implemented prototype system, using Debian 5.0 as the base environment, consists of a modified Linux kernel which restricts updates to designated file-system objects, a modified Debian package manager, and a user-space daemon (called `configd`) which is responsible for protecting an application's file-system objects from being modified by other

*Version: April 16, 2010.

applications. A control point made available in `configd` allows each configuration related file-system modification request to be examined, and either authorized or denied. As we explain in detail later, the prototype successfully prevented installation of current rootkit malware while having an imperceptible overhead to the end-user. While our discussion and prototype focus on Linux, we believe the approach can be adapted to Windows, Mac OS X, BSD, and other operating systems. Indeed, `configd` implemented on Windows could also protect the Windows registry (since it is stored on disk).

In Section 2, we give background which motivates our work. Section 3 discusses several currently-deployed alternative approaches for encapsulating an application's file-system objects on disk. Section 4 discusses design constraints and introduces the two levels which together protect system configuration and applications. Section 5 discusses our proof of concept prototype. Section 6 discusses related work. We conclude in Section 7.

2. BACKGROUND ON INSTALLERS

Our approach of limiting abuse of root privilege as it relates to configuration changes of designated file-system objects is most relevant to the case of software installs, upgrades, and uninstalls. For context and to highlight the problem, we review existing approaches to installing software on a desktop.

2.1 Application installers

The most common approach to installing applications on commodity desktops and servers is through the use of an application installer. The installer is a binary or script, often written by the same company or individual that developed the application. Its purpose, when run, is to place the various file-system objects associated with the application in the correct location and configure any system parameters (e.g., in some cases to ensure the program gets run during boot). Application installers are typically given complete control over the system during install, with users encouraged to run them with full permissions as shown in Figure 1. Whenever an application installer is run on a typical system, the entire system is opened up for modification by the installer. If the installer is malicious, or can be compromised [7], the entire system can become easily compromised. This approach does not prevent one application's installer from modifying the file-system objects of another application.

Application install scripts, like those executed through the `make install` command on many open source projects, are a slight variation of the application installer. When told to install to a user's home directory, they do not require administrator privileges. They still require administrator privileges, however, when attempting to install to a location controlled by the administrator. Using `make install` does not prevent modification of file-system objects belonging to other applications installed by the same user. While Windows encourages following the principle of each application installing into its own directory on the file-system, the practice of running application installers as root leaves the principle unenforced.

2.2 Application packages

Package managers are typically provided by an operating system (OS) or OS developer to ease development of an application installer for the platform. Instead of writing an application installer from scratch, the application developer creates a package using the package manager APIs and following the rules set by those who developed the package manager. Typically, the package consists of data used by the package manager, as well as files to be installed and scripts to run as part of the install. Common install operations are taken care of by the package manager. While the development of package managers has resulted in installers transitioning from being executables to packages (e.g., various Linux packages [10], Microsoft Installer packages [33], and Apple application packages [47]), most of these package managers still allow for executing arbitrary binary code or scripts contained in the package being installed, resulting in the same level of risk to the system as if an executable was run to perform the install. If root permission is requested by the package (or required by the package manager, as is the case with Linux packages), and the end-user enters an appropriate password when prompted (as users are now well trained to do so upon request), these scripts are run as root. The use of application packages does not prevent an application from modifying arbitrary file-system objects. One example of a malicious Debian package was a screensaver posted on `http://gnome-look.org`. The package, when installed, would also download and install a bot onto the local machine [52, 54].

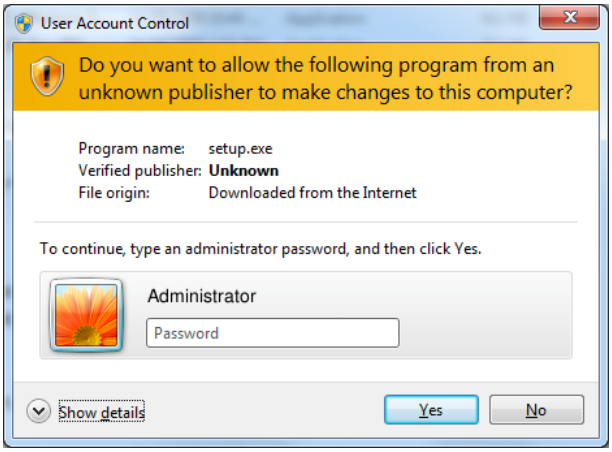

Figure 1: A Windows 7 prompt to run an installer with administrator privileges.

2.3 Apple bundles and packages

With Mac OS X, Apple introduced a new method for installing commodity applications other than application packages: application bundles. This results in two approaches on Mac OS X for installing software.

Application bundles. Application bundles are similar to packages as discussed above with two key differences. The first is that all file-system objects in the bundle will be installed into the same sub-directory by the OS (akin to the approach used on Android described in Section 3.2). The second is that scripts are not run by the OS during install. Typically, application bundles are installed through a drag-and-drop style copy operation. Commonly distributed as disk images, application bundles greatly increase the security of the system against malicious application installers.

Unfortunately, Apple still supports a second approach for installing applications and relies on the software developer to choose between them. The malicious developer is unlikely to distribute his software as a bundle when distributing a package (see below) is possible. Legitimate applications are distributed as both bundles (e.g., Mozilla Firefox) and packages (e.g., Adobe Flash).

While application bundles remove the ability for many applications to obtain root privileges, they do not mitigate the entire threat. Any application which obtains root privileges (maliciously or otherwise), even if installed as a bundle, can still modify any file on the system.

Application packages. For more complex applications, additional actions (other than simply copying the files into the application directory) may need to be performed by the installer during the install process. For these applications, Apple provides an application package framework as discussed in Section 2.2.

The status quo across multiple operating systems is to "restrict" installers by requiring the user authorize an installation by entering an administrator password before the installer can run. Once the installer is given "blind" full permission to the system, the user must trust that the installer does not abuse its privilege (since it is difficult to tell what the installer actually does with its root privilege – thus we call it blind trust).

3. APPROACHES FOR ENCAPSULATING APPLICATIONS

In this section, we discuss several approaches for encapsulating applications, using the definition of encapsulation from page 1. While we choose to implement the GoboLinux approach in `configd` (as discussed later in this paper), the alternative approaches discussed in this section are equally viable. We summarize the three approaches in Table 1. Note that although Table 1 indicates that for GoboLinux, application upgrades are not supported, the use of the GoboLinux approach in `configd` is done in such a way that application upgrades are supported.

3.1 GoboLinux

In GoboLinux [35, 36], each application is installed into its own directory (similar to Android). The actual install of a program is done through calling three scripts. `PrepareProgram` is responsible for creating the base directory where the program will be installed. `CompileProgram` takes a compressed archive of the program source, configures it with the appropriate flags so it will be installed into the directory prepared for it by `PrepareProgram`, compiles the program, and installs it. `SymlinkProgram` creates symbolic links to the various program binaries, libraries, and settings. For backwards compatibility, the common Unix directories (e.g., `/usr/bin`, `/sbin`, and `/lib`) are also linked to `/System/Links`, which is in turn populated with symbolic links for each of the applications that have been installed. To prevent an application from escaping its assigned application directory, the `make install` command is run under a special user ID [34]. This user is only allowed to modify files under two directories: the one in which the application is being installed to, and the one it was compiled from [35, 36]. In restricting an application during both compile and install, the other applications on the system remain protected against modification.

For those applications which are not complete programs in themselves but instead extensions to other applications which are already on the system (e.g., the PHP module is often installed as an extension to the web server Apache), the base application needs to be made aware of the extension. In GoboLinux, the configuration of each application is stored in the shared tree `/System/Settings/`. All files in this directory are symbolic links that point back to the settings folder, which is a sub-directory of where the application was installed. The base application provides a directory under `/System/Settings` where a module can register itself (through `SymlinkProgram`). Many distributions other than GoboLinux have also adopted this as the method of installing extensions into a base application (although the exact path to the configuration will change). Another related example involves services which should be started on system boot. On Debian based distributions, the accepted location for a script responsible for starting a service is in the directory `/etc/init.d`. GoboLinux places scripts responsible for starting the various system services in `/System/Links/Tasks`.

GoboLinux depends heavily on symbolic links being placed into the above mentioned standard directories for extensions to applications. The base GoboLinux executable `SymlinkProgram` is responsible for updating this directory tree based on the layout of files in any particular application directory (GoboLinux does not allow application installers to directly modify the symbolic links in shared directories).

A limitation of GoboLinux is that it does not cleanly support upgrades (or security patches) to an application. Each upgrade is treated as an install of a new version of the application, resulting in each version being installed into its own directory on disk. The job of sorting out which version of an application should be used by default on a system is left to `SymlinkProgram` (the PATH environment variable is set to point to `/System/Links/Executables`, a directory maintained by `SymlinkProgram`). The sharing of configuration files between different versions of an application is left up to the individual application. Indeed, each version of an application has its own copy of the configuration files stored in a `Settings` sub-directory of the application directory, alongside the various sub-directories for each version of the application which is installed.

3.2 Android

On the Android platform [1, 17], each application package is assigned its own directory and unique user id. While Android uses the Linux kernel as its base, being a single user platform, Android remapped the traditional user accounts to restrict communication between applications. The Android application installer ensures that each application is restricted to making file-system modifications in the directory it was installed into. Unlike the file-system hierarchy standard [40] as used on Linux, there are no shared directories for storing binaries, libraries, configuration files, and other elements. Android benefits greatly from the ability to mandate a file-system layout which restricts each application to single sub-directory on the file-system.

The Android platform only allows a new version of an application to be installed over top of the old if all public keys in the new version are also contained in the old version already installed (new keys cannot currently be introduced

Attribute	GoboLinux	Android	Apple App Store
Application Upgrades	Not Supported	Supported	Supported
Allows Install Scripts	Yes	No	No
Application Deployment	Unrestricted	Unrestricted	Restricted

Table 1: Characteristics of current systems which encapsulate applications on disk.

during an upgrade). During the install of an application, the application itself is not given a chance to run any installer scripts as administrator – greatly restricting the damage a particular application can do to files belonging to other applications. The platform does a good job of preventing one application from modifying another's files.

With additional work, the Android approach can be adapted to the standard Linux file-system hierarchy [40]. Instead of storing all files related to an application in a single directory, a database could be maintained which maps each individual file to the application it is associated with (the Debian package manager already keeps such information), as well as a list of public keys which are used to verify the next version of the package. The Android approach does not support scripts being run as part of the installation process (similar to Apple bundles not supporting scripts; see Section 2.3). Combining the Android approach with GoboLinux, however, allows the execution of installation scripts which can modify the configuration of the application being installed while still preventing other applications from being modified.

3.3 Apple Application Bundles

While the general application bundle is discussed in Section 2.3, a number of restrictions were made by Apple for bundled applications targeting iPhones (up to those released in January 2010) [3]. The biggest change is that each application installed onto the iPhone is limited to making file-system modifications only in the directory it was installed into. This includes limiting an application to reading and writing data files to an application specific area of the file-system (similar to Android).

In contrast to Android, Apple application bundles targeted to the iPhone are not signed with the key of the software developer. Instead, each application must be signed by Apple before it can be run on an iPhone (we ignore "jailbroken" iPhones in our discussion). Before signing an iPhone application bundle, Apple examines the application to ensure it meets their criteria [2]. For application bundles designed for Mac OS X, Apple has no such restriction that the bundle be verified and signed by Apple before it can be installed.

4. DESIGN FOR CONTROLLING ROOT PRIVILEGES ON FILE SYSTEMS

Our main objective is to reduce root privilege so that programs, such as installers, cannot take advantage of overly coarse access controls to abuse the privileges they have been given. The design of our approach is subject to several self-imposed constraints. We believe that, to be viable, any alternate approach for restricting file-system privileges on the desktop at a per-application level would need to fulfil the following considerations.

1. Compatibility with current file-system layouts. In designing a Linux-based prototype of the proposal, our goal was to avoid requiring redesign of the current file-system hierarchy [40] in favour of a solution compatible with the current file-system layout. Applications are protected against modification while retaining the current file-system layout – installing files to directories shared with other applications. In contrast, GoboLinux [36] and Apple (in Mac OS X) did redesign the file-system hierarchy. Their motivations were apparently to impose cleanliness in restricting each application to its own directory. While the separation of each application into its own directory may simplify the challenge of restricting configuration changes on a per-application basis, a backwards compatibility layer is still required to support applications not designed for the new layout.

2. Minimal impact on day-to-day operations. Most of the time, a computer is used to perform day-to-day tasks (run applications) with a constant configuration of the applications and operating system. Occasionally, its configuration is modified in order to expand/modify the tasks it can perform (e.g., applications are installed, updated, removed, or reconfigured). Our proposal (and indeed any alternate `configd` approach) should impose no noticeable impact on such day-to-day operations, with no changes to regular user work flow.

3. Backwards compatibility for current installers. We introduce new restrictions on an application's ability to modify file-system objects. These restrictions will typically influence the install, upgrade, and removal of applications. It is unrealistic to assume that all application packages will be modified in parallel during deployment of such a solution. Backwards compatibility is therefore critical for incremental deployability. Our prototype did not change the Debian package structure at all, maintaining backwards compatibility with versions of `dpkg` not designed to work with `configd`.

4. Usability. Our focus in this paper is on providing a solution which can be used by non-expert users, and to avoid forcing upon users choices which they are ill-equipped to respond to correctly. Our solution achieves this goal, allowing an applications' file-system objects to be protected against modification during install, upgrade, uninstall, and at runtime, without presenting the user with complex choices. While our prototype solution did leave enabled the option of querying the user about file-system operations, this feature can be safely disabled (as discussed in Section 5.5).

5. Other considerations. We assume that the user of a computer system can be trusted to not be malicious. The proposal, therefore, does not protect against physical attacks, such as rebooting into a kernel that does not enforce the proposed protection mechanism.[1] Its security also

[1]While some have proposed that the user cannot be trusted [55, 28], our work avoids declaring the user as the enemy and preventing them from modifying their own system. We favour persuasive tactics as a tool to encourage users to properly maintain their system while not taking the control out of their hands.

assumes that applications cannot obtain kernel level control of the system,[2] although on most current systems, any application running with root access can modify the running kernel. This assumption therefore relies on the proper functioning of a subset of mechanisms designed to protect kernel data and code against compromise by applications running with root access, as proposed in the considerable independent literature [4, 38, 57, 27, 16, 39, 26, 58, 41, 11, 15]).

4.1 A control point for division of root privileges

To build a system designed to reduce root abuse of file-system privileges, we first separate configuration related activities (those configuration actions affecting the applications installed or their global configuration as stored on disk). We then further subdivide the configuration privilege to remove the ability of an application installer to modify any file-system object other than those which are part of the application being installed (or upgraded/removed).

Our prototype design consists of two main elements: a kernel extension and a user-space daemon. The user-space daemon is responsible for the bulk of the work, namely, ensuring that one application cannot modify files related to a different application. The kernel is responsible for denying (or forwarding) requests to modify protected *file-system objects*, by which we mean files (including binaries), directories, symbolic links, and other objects that are part of the file-system. Protected file-system objects (which we call *c-locked* file-system objects, short for *configuration locked*) are designated (marked) as such by the user-space daemon (an alternate method of protecting file-system objects is by using a union file-system to redirect writes [60]). Any application file-system object marked as part of the system configuration (and hence to be protected against modification by other applications) must be so designated. While we leave open the exact set of c-locked file-system objects, we view the set to include shared libraries, executables, system configuration files, start-up scripts, and other file-system objects which do not change as a result of day-to-day system use. In effect, the system configuration on disk includes the set of applications installed as well as each application's files which are required at run time. Only a process holding a newly introduced *configuration permission* is allowed to modify the system configuration on disk (and hence c-locked file-system objects).

The exact distribution of duties within our design has the kernel responsible for:

1. Restricting to programs running with configuration permission the ability to delete, move, and write to c-locked file-system objects. By design, "root" is not allowed to make arbitrary changes to c-locked file-system objects (including the kernel image) – changes are limited to processes running with configuration privilege.
2. Restricting the ability to obtain configuration permission. In our prototype, this is done by allowing only a single process, the configuration daemon (`configd`) to have configuration permission.
3. Restricting the ability to control the process running with configuration permission (e.g., by not allowing `configd` to be killed or modified by a debugger).

In restricting configuration permission to a single daemon, we introduce a chokepoint within which we can further subdivide file-system access by application.[3] We perform the actual subdivision in the above-mentioned configuration daemon `configd`. It performs the following operations:

1. Respond to requests for configuration changes from processes running on the system. In our prototype, requests were at the granularity of package installs and uninstalls, but our design could be easily modified to handle other granularities.
2. Designate file-system objects as c-locked. It marks a setting in a data structure associated with the object to denote this designation. In our prototype, each file installed when upgrading or installing a package is marked as c-locked.
3. Perform authorized changes to the configuration of the system.

We now discuss in more detail how the two elements work together to restrict the ability for an application to modify file-system objects belonging to another application.

4.2 Linux kernel protection of c-locked file-system objects

How a kernel handles file-system objects can directly affect the security of c-locking. In the Linux kernel, the key file-system data structures directly related to the protection of file-system objects by `configd` are the *directory entry* (or dentry) and inode [44]. The inode data structure contains most of the information related to the file data and meta data (e.g., the traditional Unix file-system permissions read, write, and execute) and pointers to the actual blocks on disk which hold the file contents. The dentry contains information related to the specific instance of a file in a particular directory, including the name of the file (as it appears in that directory) and a pointer to the inode. For the purposes of c-locking, the dentry inherits the c-locked status of the inode. If the inode is marked as c-locked, then the directory entry can be deleted or moved only by `configd`. File operations on an inode which is not c-locked are restricted through current access-control restrictions (including traditional Unix file permissions). Figure 2 demonstrates the relationship between inodes and dentries.

Symbolic Links. Symbolic links are directory entries in Linux pointing to an inode containing a path string. When opening a symbolic link, the kernel retrieves the path name from the symbolic link inode. It then follows the retrieved path name to obtain another dentry and inode (which is either yet another symbolic link or some other element such as a file or directory). The proposed system supports either c-locking the symbolic link, the object it points to, or both.

Hard Links. A hard link is a directory entry in Linux pointing to the same inode as another directory entry. As with regular files, because the inode itself contains the c-lock flag, any hard link pointing to the inode inherits the c-locked attribute associated with the inode. An attacker

[2]We define the kernel (and hence kernel level control) to include only those aspects running with elevated CPU privileges (ring 0 privileges on x86); this definition does not include core system libraries installed alongside the OS but run in user-space.

[3]While it is possible to use the kernel as the chokepoint, our preliminary exploration in this direction suggested that implementing the functionality required to further subdivide root on a per-application basis directly in the kernel introduces functionality into the kernel which is already available in user-space.

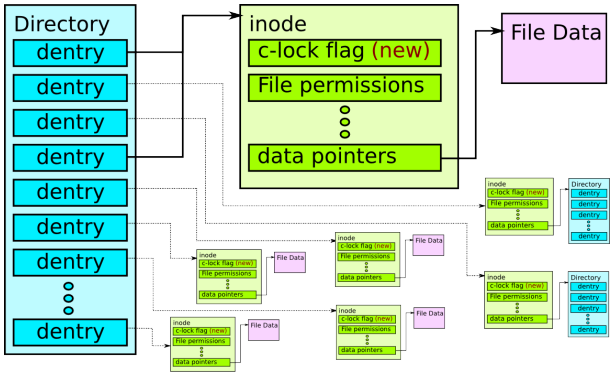

Figure 2: File-system data-structure layout including new c-lock flag.

does not gain modification privilege by creating a hard link to a file-system object protected by c-locking. The ability to create a hard link to a c-locked file is restricted, being either allowed or denied by `configd`.

Directories. A directory is an inode which instead of pointing to file data, points to a list of dentries. While previous approaches [9, 63] focused on protecting files more than directories, there are cases in which a directory should be protected. As an example, during start-up the Debian `/etc/rcS.d` directory is accessed and every file (or file pointed to by a symbolic link) in this directory is run. Any malware installed into this directory would be started automatically during system boot. The proposed system can protect directories since they can be c-locked in the same manner as files and symbolic links.

4.3 `configd`

The prototype `configd` is designed to subdivide root file-system permissions on a per-application basis. In our framework, `configd`, or its equivalent, becomes a chokepoint which applications must use in order to modify c-locked file-system objects (and hence the configuration of the system). To enforce that `configd` is the only way that c-locked file-system objects can be modified, the kernel grants the new configuration permission to `configd` alone. By delegating this privilege to `configd`, the kernel need not know about every application on the system or what file belongs to which application; it need only recognize that a specific file is c-locked and leave the handling of this file to `configd`.

`configd` must be started early during the boot process (`configd` itself restricts changes to the boot process). Once `configd` has started, other programs are prevented by the OS kernel from obtaining configuration permission. The design of `configd` takes advantage of the temporal nature of software installs. At the time of installation of application software, it is assumed that the operating system and `configd` are already installed and running. This assumption is reasonable if `configd` is made part of the OS or core system.

4.3.1 *Example* `configd` *rule set*

While the core `configd` approach can use any number of different protection mechanisms for separating files on a per-application basis, we chose to depend on Debian packages (and indeed, the package manager – `dpkg`) in our solution. During the package install process, we allow the following actions to be performed.

1. If A is a c-locked file-system object whose contents under a cryptographic hash have the same value as *A.dpkg-new*, then *A.dpkg-new* can replace A (i.e., if A and *A.dpkg-new* contain the same data). We do not mandate any specific cryptographic hash algorithm, other than to stipulate that, at minimum, it must have second pre-image resistance [30] (our `configd` prototype currently supports SHA-1 and can be easily expanded to support others).
2. If A is a c-locked file containing one or more public keys and *A.dpkg-new* is another file containing a digital signature verified by using a public key in A, then *A.dpkg-new* is allowed to replace A.
3. If c-locked file A is associated with package *PKG* and is not associated with any other package installed on the system, then when upgrading package *PKG*, A may be modified.
4. If c-locked file A is associated with package *PKG* and is not associated with any other package installed on the system, then install scripts associated with package *PKG* may modify A.
5. All other operations involving a c-locked file are considered to be potentially dangerous. They can either be presented to an expert user for additional oversight, or simply denied (as discussed in Section 5.5).

Of the above rules, some merit further discussion. Rule 2 adapts concepts used by the Android OS [1], but to the file (as discussed by Wurster et al. [61]) as opposed to package level. The rule relies upon the use of public keys and signatures, but does not rely on a PKI.

Rule 3 allows the modification of all files associated with a package when a new version of the package is installed. The rule requires that `configd` keep track of packages installed on the system, as well as which files are associated with which packages. We discuss the semantics of how our prototype handled packages in Section 5.3. It is important to note, however, that a package cannot modify any file on the system simply by asserting ownership of the file (through including the file in the list of files the package is associated with). Any file which is listed as belonging to more than one package cannot be arbitrarily modified by any package. The option still exists, however, for a file associated with more than one package to be updated through rule 2.

Rule 4 restricts, through exclusion, the files that an install script can modify. The rule is borrowed from GoboLinux [36] and discussed in more detail in Section 3.1.

Our testing confirmed that the above rule set allows package installs and upgrades to be automatically allowed, while providing encapsulation. While the option of querying the user with remaining operations was left enabled in the prototype, we found in testing that during upgrades, the user is not queried at all – with the exception of updating global configuration files (see Section 5.4), rules 1 through 4 accomodated all install operations.

5. PROTOTYPE IMPLEMENTATION AND EVALUATION

5.1 Kernel extensions

To support c-locked file-system objects, we used the extended attributes functionality [50] of file-systems such as

ext3 and XFS. This is the same approach used by SELinux [42]. In so doing, the underlying file-system-specific data structures do not need to be modified. The extended attributes are tied to the inode. We used the *trusted* extended attribute name space because it supported setting extended attributes on symbolic links. We created our c-locking protection mechanism as a Linux Security Module [29, 59]. The kernel implementation was approximately 2200 lines of code, including the backward compatibility layer. A new device node (`/dev/configd`) was used as the interface between the user-space `configd` and the modified kernel, allowing communication between the two. The process of opening the device node initiated c-locking protection in the modified kernel. The kernel understands and responds to commands sent by the user-space `configd` through the new device node, allowing `configd` to disable raw disk writes (as discussed in Section 5.2), and mark files as c-locked (as discussed in Section 4.1).

To prevent applications from being able to modify c-locked file-system objects through modification of the kernel, access to both `/dev/kmem` and `/dev/mem` was restricted (both are options built into the Linux kernel).

5.2 `configd` implementation

While the primary purpose of `configd` is to encapsulate applications on disk, during the course of prototype development several additional features were added. These include:

1. A mounting module which can decide whether a request to mount a file-system should be performed (it performs the operation, if allowed). To ensure that c-locked file-system objects remain accessible by applications they are associated with, we must ensure that a new file-system cannot be mounted over top of c-locked file-system objects. `configd` does this by examining the *trusted.configd.nomount* file-system extended attribute. For directories which should not be mounted over (e.g. `/usr`), this extended attribute should be set on the directory inode. The mounting module also informs the kernel through the `configd` device node that requests to write to the underlying raw hard drive sectors should be denied, since allowing such requests would undermine the security of both c-locking and per-application restrictions enforced by `configd`.

2. A modprobe module which handles requests for the insertion or removal of kernel modules. In order to ensure that `configd` remains the chokepoint for restricting file-system objects on a per-application basis, root must not be allowed to install arbitrary code into the running kernel. While the prototype version of this module currently accepts all requests, kernel module loading can be easily restricted based on a number of criteria, including whether the module is signed with a recognizable key [26] (such an approach would not require a complete PKI infrastructure).

5.3 Debian package manager modifications

Because the Debian package manager (`dpkg`) is responsible for performing most configuration changes on a system, we integrated `dpkg` with `configd`. While other methods of restricting configuration changes at a per-application level may choose to totally replace `dpkg`, augmenting `dpkg` to communicate with `configd` was suitable for our prototype (`configd` had the final say as to whether all requests for operations on c-locked files are allowed). The modified `dpkg` informed `configd` about each package which was being installed or upgraded, and also marked as c-locked each file installed when upgrading or installing a package, regardless of whether the file was previously c-locked. Because it is considered an error on Debian to have a file belonging to two unrelated packages [20], Debian's package approach lends itself nicely to a clean separation between applications. Since packages can contain arbitrary files, any file which should be c-locked either already is, or can be, distributed in a package.

In our prototype, the Debian package manager and surrounding infrastructure was responsible for preventing one application from assuming the name of another (and hence being able to modify the files associated with the second package). Such an approach depends on the security of the packaging system in Debian, which although reasonably secure against intrusion, is not perfect [10]. To reduce the dependence on Debian to keep packages from replacing other un-related packages in the archive, an approach similar to that used in Android could be deployed. In Android, a form of digital signature not tied to an identity is used as a way of restricting replacement of packages (see Section 3.2). A package is typically signed with a private key held by the developer. Any new version of that package is allowed to replace the installed one if it is signed with a private key verifiable with the corresponding public key contained in the currently installed package. In this way, application updates are restricted to those software authors holding the private signing key. Unless two applications are written by the same developer, and assuming that private keys are not generally shared between developers, the two applications are unlikely to have any identical keys and hence will not be able to modify each other. In adapting the approach to Debian packages, we can eliminate the risks associated with relying on the Debian packaging team to properly keep packages distinct [10].

5.4 Allowing Scripts During Package Install and Uninstall

In Section 4.3.1, rule 4 states that if c-locked file A is associated with package PKG and is not associated with any other package installed on the system, then install scripts associated with package PKG may modify A. In order to enforce that an install script contained in a package may only modify files on the system associated with that package (and no other package), we implemented an ability to run install scripts within `configd`. The approach closely parallels the approach used by GoboLinux for installing applications (see Section 3.1). The following procedure was followed for running both install and uninstall scripts:

1. An unused user ID (*UID*) is allocated by `configd`. While `configd` examined `/etc/passwd` in our prototype, different methods may be required when using alternate approaches for user account control.

2. For each file associated with package *PKG* (and not associated with any other package installed on the system), the owner of the file is recorded by `configd` and then changed to *UID*.

3. The script is run as user *UID* by `configd`. In running the script as a user who only has permission to modify files associated with the package, other applications

on the system are protected by the standard access controls on the system.

4. For each file associated with the package *PKG*, the *UID* is changed back to the value stored in step 2, unless the user owning the file has been changed by the script.

5. The *UID* allocated in step 1 is freed.

The approach allows an (un)install script associated with package *PKG* to modify any file associated with the same package, but does not allow the install script to modify any file associated with any other package installed on the system. It achieves the design goal of restricting file-system modifications such that one application can not modify the file-system objects associated with another application installed on the system. Indeed, simply implementing the approach into `dpkg` without fully implementing `configd` has benefits over the current approach of allowing unrestricted access to the file-system by install scripts.

While we did not encounter any scripts which were affected by the change from running as root to running as an unprivileged *UID*, if such a scenario did pose problems, the use of a fake root would allow `configd` to better emulate the permissions an install script has without `configd` running. `fakeroot` is a program commonly used in the Linux environment to make an application believe it has root privileges. The operations performed while running under `fakeroot` are recorded with the goal of being able to save and replay them later. This approach is most often used when creating a distribution package from a source archive, but can also be used by `configd`. It is also the approach used in related work to record the actions performed by an installer [53, 48].

5.5 Handling of Operations not Automatically Allowed

While the prototype left enabled the option of querying the user for operations not allowed by other rules discussed in Section 4.3.1, we believe that this rule can be disabled when deploying `configd` to non-expert user systems, causing `configd` to reject of any file-system operation not allowed by the other rules. During testing, we were not queried by `configd` at all about modifications to the file-system during package install and upgrade. The user continues to be queried by `configd` about configuration file changes performed manually by root. To prevent malware from interfering with questions `configd` poses to the user, we extended the kernel, preventing other applications from running while the user is being queried.

Because root no longer has permission to modify arbitrary files on disk, any configuration changes performed directly by the root user will be potentially be disallowed by `configd` or a related per-application encapsulation mechanism (configuration changes performed during install by the related install script are easily allowed; see Section 3). In the prototype system, configuration changes performed by the physical user acting as root ended up being approved, since the person modifying the application configuration and the person approving the change when queried by `configd` are one and the same. Because any updates to configuration require an individual acting as root to perform them, we believe the extra step of the same individual verifying the change before it is propagated to disk to be minor (e.g., the user updates the configuration file and then authorizes that the update be written to disk when prompted by `configd`). Indeed, we can avoid querying the user about configuration changes if the implementation of `configd` exports a user interface for performing changes (e.g., by providing a text editor). Such an ability does not detract from the security of the system because applications still do not gain the ability to write to file-system objects belonging to other applications.

5.6 Performance evaluation

To test the performance of our kernel modifications on file-system intensive day-to-day operations, we performed a complete compile of the Linux 2.6.31.5 kernel. We unpacked, configured (`make allmodconfig`), compiled, and removed the directory tree containing the compiled kernel. We chose a kernel unzip, compile, and removal because of the number of required disk operations, heavily exercising the file-system as well as our prototype c-locking Linux security module. We ran the test on two different 2.6.28.7 Linux kernels. The first test was with c-locking support not compiled in, and averaged 158 minutes and 52 seconds over three runs. The second timing was performed with c-locking enabled and `configd` running, and averaged 166 minutes and 34 seconds over three runs. Both tests were run on the same Pentium 4 2.8GHz with 1Gb of RAM. Over the three test runs, the average increased run time for the compile with c-locking enforcement enabled was 4.8%. For day to day operations which do not involve heavy file-system activity, we expect the overhead of `configd` to be well under 4.8%. We also expect alternate implementations of `configd` functionality would produce comparable results.

5.7 Verification that application encapsulation is enforced

Malware frequently modifies file-system objects not directly associated with the malware itself (e.g., replacing *ls*). This provides an appropriate test case for the proposed restriction of configuration changes on a per-application basis during install. Under the new root file-system restrictions, the test is to verify that applications (malicious or other) running with root privileges cannot modify other applications file-system objects.

To test how well the mechanism presented in this paper protect a system when exposed to malware, we became root on a system with `configd` running and kernel protections enabled. We then attempted to run six different Linux rootkit installers.[4] Linux rootkits can be grouped into two categories: those that use some method to gain access to kernel memory, installing themselves in the running kernel and then operating at kernel level, hiding their actions from even root processes; and rootkits that replace core system binaries that are often used by the root user in examining a system. Using the six representative rootkits, we confirmed that the installer failed to gain access to the kernel because of disabled write access to `/dev/kmem` (which would otherwise undermine `configd`), and that `configd` works as expected (i.e., file-system changes possible by malware are restricted to prevent other applications from being modified). That the rootkits failed to gain access to the kernel was verified by examining errors returned by the rootkit installer when attempting the install. The integrity of other applications

[4]All Linux rootkits tested were from `http://packetstormsecurity.org/UNIX/penetration/rootkits/`

file-system objects was verified through comparing cryptographic hashes using Tripwire [23].

We selected six representative Linux rootkits, two that modify the kernel and four that replace system binaries. Both kernel-based rootkits (`suckit2` and `mood-nt`) failed to install because of disabled write access to `/dev/kmem` in the prototype's modified kernel. The `mood-nt` kernel based rootkit which we tested also attempted to replace `/bin/init`. The attempt was denied by the prototype[5] because `/bin/init` is part of a different application on the system. The four binary replacement rootkits (`ARK 1.0.1`, `cb-r00tkit`, `dica`, and Linux Rootkit 5) all resulted in file-system operations which were denied because they attempted to either replace or delete core system binaries (e.g., `ls`, `netstat`, `top`, and `ps`). The core system binaries installed belong to applications other than the rootkits and hence changes to them by the rootkit installer were disallowed by our prototype.

6. RELATED WORK

Here we discuss selected related work, beyond that discussed in Section 2 and Section 3.

6.1 Secure Software Installation

Venkatakrishnan et al. [53] proposed RPMShield, a system where actions which will be performed during install of a package are presented to the administrator for verification and then all install actions are logged. RPMShield concentrates on install time, not preventing already-installed applications from modifying the system if they are run as root. While `configd` focuses on encapsulating an application's file-system objects, RPMShield focuses on allowing the system administrator to examine and approve the actions which will be performed during install.

Kato et al. [21] proposed SoftwarePot, an approach where each application is encapsulated in its own sandbox, with mediated access to the file-system and network. Shared files are accessed by mapping sandbox-specific file-names to global file-system objects. The mapping between sandbox-specific files and global file-system objects requires additional information not currently distributed with an application package. SoftwarePot encountered a 21% overhead, while `configd` encountered a 4.8% overhead.

Sun et al. [48] proposed grouping applications into two categories, those which are trusted, and those which are not. All untrusted applications are installed inside a common sandbox, while trusted applications are not. The approach relies on malicious applications always being classified as untrusted. It does not prevent trusted applications from modifying file-system objects related to other trusted applications (and indeed, untrusted applications can modify file-system objects related to other untrusted applications). `configd`, in contrast, does not distinguish between trusted and untrusted applications, treating all applications equal and restricting the modifications which can be performed on file-system objects.

6.2 Rootkit resistant disks

Butler et al. [9] 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 the file-system. Blocks on disk become marked as associated with a USB key when they are updated while the key is installed. In their approach, the user is involved in differentiating between when a system should be used to perform day-to-day operations and when the system is being configured. This separation, however, does not carry over into isolating day-to-day and configuration operations. Because the protection mechanism is implemented underneath the operating system at the block level, applications used for performing day-to-day operations continue to run (and even inherit configuration permission) when the user inserts a USB key indicating they want to change the configuration of the system. The Butler et al. approach of attempting to restrict configuration operations closely parallels the first step in our limiting the potential for abuse in root file-system privilege – the separation of configuration from normal day-to-day activities performed on a system.

The approach taken by Butler et al. attempts to further minimize the potential for abuse through the use of multiple USB keys, but does not reach an application level granularity. Indeed, they suggest using different tokens for different roles (e.g., one token associated with all binaries and another associated with all configuration files) [9].

6.3 Other related work

The splitting of root privilege is a common technique for limiting abuse in areas other than file-system control. Techniques such as capabilities [24, 18] and fine-grained access control [29] also split up root, but focus mainly on installed applications, not the installers themselves.

On the iPhone platform, each application must be signed by Apple before being loaded onto the device. This approach assumes Apple will be able to properly vet all applications before allowing them to be loaded onto the device. Apple also has a mechanism for disabling malicious applications which happen to slip through [22]. The approach has scalability drawbacks in that it relies on a central authority to certify all software for the platform.

In the past few years, virtual machines (VMs) have started to become much more popular in server environments, to allow a single machine to run multiple instances of an operating system [5]. As part of the popularization of virtual environments, the opportunity to introduce additional segregation between applications has arisen. Each VM instance runs its own instance of an operating system and is assigned its own file-system and display. A typical setup involving virtual machines still groups many related applications together in a single VM. In this paper, we focus on a method for dividing up root file-system privilege to prevent abuse by applications running on the same instance of the operating system, regardless of whether or not that OS happens to be running in a virtual machine. While the practice of an ordinary user installing applications into their home directory avoids root entirely, the application's file-system objects are not protected against modification by other applications the user installs (or indeed, any application the user happens to run).

Ioannidis et al. [19] introduced the concept of sub-operating systems, marking each file with a label indicating where it

[5]In our prototype, the denial was performed by the prototype user, but would be performed automatically if the user-interface is disabled. The rootkit install was not able to put up a fake `configd` prompt because of protections discussed in Section 5.5.

came from. These labels restrict what data files an application is allowed to access at a finer granularity than the user. Sub-OS does not explicitly tackle limiting the abuse of root file-system privilege during the process of installing, upgrading, and removing an application. Polaris [45] likewise focuses on application data, restricting what user files an application can access based on user interaction with the window manager.

Fine-grained access control systems, such as SELinux [29] and those implemented by Solaris [8], restrict an application's permission based on the labels assigned to both that application and the resources it wishes to use. These systems have the potential to split up root file-system privilege, parallelling the approach used herein. Traditionally, however, such policies have focused on run time system state (i.e., when the system is being used for day-to-day activities) as opposed to installers and related file-system configuration operations. In Linux, the default SELinux security policy has `dpkg` being granted write access to almost every file on the system [12]. Other systems, such as AppArmor [6], appear to work best when new applications are not being installed or upgraded. In the current environment where applications are routinely upgraded, not supporting installs or upgrades is a problem.

While projects such as DTE-enhanced UNIX [56] and XENIX [49] restrict the privileges of root (including root's ability to configure the system), it seems most installs on these systems still happen with full privileges (`admin_d` privilege in DTE, TSP in XENIX), having full access to all file-system objects on disk. For systems using the OpenBSD `schg` [25] and ext2 immutable [51] flags, any application can be given the ability to change an immutable file – the user can simply be asked to run an application after acquiring sufficient configuration privileges. SVFS [63] protects files on disk but is susceptible to the same problem of inadequate control over installation applications.

There have been many attempts to detect malicious modifications to system configuration. Windows file protection (WFP) [13, 31] maintains a database of specific files which are protected, along with digital signatures of them. WFP is designed, however, to protect against a non-malicious end-user, preventing only accidental system modification. Pennington et al. [37] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. Strunk et al. [46] proposed logging all file modifications for a period of time to assist in the recovery after malware infection. Tripwire [23] maintains cryptographic hashes of all files in an attempt to detect modifications. All these proposals detect modifications after the fact. Applications such as registry watchers [43] and clean uninstallers [32] attempt to either detect or revert changes made to a system by an application installer. These systems similarly don't prevent changes in system configuration.

The separation of configuration privileges as proposed in this paper prevents installers from making unauthorized changes to system state, leading to a proactive rather than reactive approach to limiting system configuration changes. Package managers [10] and the Microsoft installer [33] both limit system configuration actions allowed by packages designed for their system, but do not prevent applications from simply providing their own installer (or install script), bypassing the limits enforced by the package manager.

This paper follows preliminary work first introduced in a workshop paper [62]. We expand the work by evaluating current installers, introducing the concept of restricting configuration changes at a per-application granularity, protecting file-system objects (as opposed to just files), and discussing a working prototype. The prototype implementation confirms the feasibility of protecting against abuse of root file-system privileges, and is evaluated for both its performance and ability to defend against several rootkits.

7. SUMMARY

The benefits of preventing one application from modifying another's file-system objects are well-recognized in the smart phone environment. On modern desktop computing platforms which are less tightly controlled, and more generic, there has been little progress on architectural designs which support encapsulation of applications throughout frequent install, upgrade, and uninstall processes. Unix (including Linux) provides a challenge in that much of its design is from an era where applications were all installed into common directories. While Windows does slightly better by encouraging each application be installed into its own directory, the way applications are installed leaves the system open to attack.

In this paper, we present a framework for restricting configuration changes, including kernel extensions and an associated user-space daemon. The status quo on end-user operating systems is dangerous – giving every application complete control during install, upgrade, and removal. We provide the foundation of a full solution including concept, architecture, design, and prototype implementation proving out the design. The design provides a control point to reduce root abuse of file-system privileges without breaking normal software operation, and has acceptable performance. As a test case that it stops damaging file-system changes, we confirmed that file-system writes by common rootkits are trapped.

The design of our control point allows several different control and policy mechanisms. In our prototype, the policy mechanism exposed an interface suitable for an expert user, but the control point can also be used for alternate mechanisms such as those used by Android and GoboLinux. Our end-goal is to eliminate the property whereby every process running with root privilege can change arbitrary files on disk, as this is commonly abused by malware on current desktop operating systems. A necessary part of this involves restricting the ability for applications to modify each other's file-system objects on disk. Our proposal mitigates the security risks associated with install mechanisms in common use today, wherein software installers (typically downloaded from the Internet) are run as root. The problem addressed herein is long-standing.

Acknowledgement

We thank the many individuals who provided feedback on preliminary drafts of this work. The second author acknowledges NSERC for an NSERC Discovery Grant and his Canada Research Chair in Internet Authentication and Computer Security. Partial funding from NSERC ISSNet is also acknowledged.

References

[1] Android developers. Web site (viewed 18 Nov 2009). http://developer.android.com.

[2] Apple, Inc. iPhone developer program license agreement, Jun 2008.

[3] Apple Inc. *iPhone Application Programming Guide*, Jan 2010. http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf.

[4] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. 2007 IEEE Symp. on Security and Privacy*, 2007.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

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

[7] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *USENIX 2006 Workshop on Hot Topics in Security (HotSec 2006)*.

[8] G. Brunette. Restricting service administration in the Solaris 10 operating system. Technical Report 819-2887-10, Sun Microsystems, 2005.

[9] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *Proc. 15th ACM CCS*, 2008.

[10] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proc. 15th ACM CCS*, 2008.

[11] 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 CCS*, 2009.

[12] R. Coker. Re: [dse-dev] refpolicy: domains need access to the apt's pty and fifos. Mailing List Posting, Mar 2008. http://www.nsa.gov/research/selinux/list-archive/0803/25307.shtml.

[13] J. Collake. Hacking Windows file protection. Web Page, 2007. http://www.bitsum.com/aboutwfp.asp.

[14] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[15] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proc. 16th ACM CCS*, 2009.

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

[17] Google. Android developer guide. Developer Website, 2009. http://developer.android.com/guide/publishing/app-signing.html.

[18] S. E. Hallyn and A. G. Morgan. Linux capabilities: Making them work. In *Proc. Ottawa Linux Symposium*, Jul 2008.

[19] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *Proc. 10th Workshop on ACM SIGOPS*, pages 108–115, 2002.

[20] I. Jackson and C. Schwarz. *Debian Policy Manual*, 1998. http://www.debian.org/doc/debian-policy/.

[21] K. Kato and Y. Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. In *Proc. of Int. Symp. on Software Security*, volume Lecture Notes in Computer Science 2609/2003, pages 217–224, 2003.

[22] A. Kim. Apple's ability to deactivate malicious app store apps. Web site, 2008. http://www.macrumors.com/2008/08/06/apples-ability-to-deactivate-malicious-app-store-apps/.

[23] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proc. 2nd ACM CCS*, 1994.

[24] A. Kjeldaas. Linux capability faq v0.1. Mailing List Posting, Aug 1998. http://lkml.indiana.edu/hypermail/linux/kernel/9808.1/0178.html.

[25] Y. Korff, P. Hope, and B. Potter. *Mastering FreeBSD and OpenBSD Security*, chapter 2.1.2. O'Reilly, 2005.

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

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

[28] Q. Liu, R. Safavi-Naini, and N. P. Sheppard. Digital rights management for content distribution. In *Proc. Australasian Information Security Workshop Conference on ACSW Frontiers*, volume 21, pages 49–58, 2003.

[29] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX '01*, 2001.

[30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[31] Microsoft. Description of the Windows file protection feature. Web Page, 2007. http://support.microsoft.com/kb/222193.

[32] Microsoft. Description of the windows installer cleanup utility. Technical Report Q290301, Microsoft, 2008. http://support.microsoft.com/kb/290301.

[33] J. Moskowitz and D. Sanoy. *The Definitive Guide to Windows Installer Technology*. Realtimepublishers.com, 2002. http://nexus.realtimepublishers.com/dgwit.php.

[34] H. Muhammad. Compiling from source. Web Page (viewed 16 Feb 2010). http://gobo.kundor.org/wiki/Compiling_From_Source.

[35] H. Muhammad. The Unix tree rethought: an introduction to GoboLinux. Kuro5hin Article, May 2003. http://gobolinux.org/index.php?page=k5.

[36] H. Muhammad and A. Detsch. Uma nova proposta para a árvore de diretórios UNIX. In *Proceedings of the III WSL - Workshop em Software Livre*, 2002.

[37] 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 Symp.*, 2003.

[38] 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 Symp.*, 2004.

[39] N. L. Petroni Jr., T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detec-

tion of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symp.*, 2006.

[40] R. Russell, D. Quinlan, and C. Yeoh. *Filesystem Hierarchy Standard*. Filesystem Hierarchy Standard Group, 2.3 edition, 2004. `http://www.pathname.com/fhs/`.

[41] M. Sharif, W. Lee, and W. Cui. Secure in-VM monitoring using hardware virtualization. In *Proc. 16th ACM CCS*, 2009.

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

[43] E. Software. Registry watch. Software Application (viewed 23 Apr 2009). `http://www.easydesksoftware.com/regwatch.htm`.

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

[45] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: virus-safe computing for Windows XP. *Communications of the ACM*, 49(9):83–88, 2006.

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

[47] S. Sudre. Packagemaker how-to. Web site (viewed 29 Oct 2009). `http://s.sudre.free.fr/Stuff/PackageMaker_Howto.html`.

[48] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding malware defense by securing software installations. *Lecture Notes in Computer Science*, 5137/2008:164–185, 2008.

[49] Trusted XENIX version 3.0 final evaluation report. Technical Report CSC-EPL-92-001, National Computer Security Center, 1992.

[50] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux ext2/ext3 filesystem. In *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–243, 2002.

[51] C. Tyler. *Fedora Linux*, chapter 8.4. O'Reilly, 2007.

[52] Social engineering (trojan) via gnome-loook.org. Web Page (viewed 13 Feb 2010). `http://ubuntuforums.org/showthread.php?t=1349801`.

[53] V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An approach for secure software installation. In *Proc. LISA '02: Eighteenth Systems Administration Conference*, 2002.

[54] K. Vervloesem. Linux malware: an incident and some solutions. LWN.net Article, Dec 2009. `http://lwn.net/Articles/367024/`.

[55] S. Vidyaraman, M. Chandrasekaran, and S. Upadhyaya. Position: The user is the enemy. In *Proc. 2007 Workshop on New Security Paradigms*, pages 75–80, 2007.

[56] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX Security Symp.*, 1996.

[57] 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)*, 2005.

[58] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. 16th ACM CCS*, 2009.

[59] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symp.*, 2002.

[60] C. P. Wright and E. Zadok. Unionfs: Bringing filesystems together. *Linux Journal*, (128):24–29, Dec 2004.

[61] G. Wurster and P. van Oorschot. Self-signed executables: Restricting replacement of program binaries by malware. In *USENIX 2007 Workshop on Hot Topics in Security (HotSec 2007)*.

[62] G. Wurster and P. C. van Oorschot. System configuration as a privilege. In *USENIX 2009 Workshop on Hot Topics in Security (HotSec 2009)*.

[63] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *Proc. Third IEEE International Security in Storage Workshop (SISW'05)*, 2005.