Enhancing the OS against Security Threats in System Administration

Nuno Santos[1], Rodrigo Rodrigues[2], and Bryan Ford[3]

[1] MPI-SWS
[2] CITI / Universidade Nova de Lisboa
[3] Yale University

Abstract. The consequences of security breaches due to system administrator errors can be catastrophic. Software systems in general, and OSes in particular, ultimately depend on a fully trusted administrator whom is granted superuser privileges that allow him to fully control the system. Consequently, an administrator acting negligently or unethically can easily compromise user data in irreversible ways by leaking, modifying, or deleting data. In this paper we propose a new set of guiding principles for OS design that we call the *broker security model*. Our model aims to increase OS security without hindering manageability. This is achieved by a two-step process that (1) restricts administrator privileges to preclude inspection and modification of user data, and (2) allows for management tasks that are mediated by a layer of trusted programs—*brokers*—interposed between the management interface and system objects. We demonstrate the viability of this approach by building BrokULOS, a Linux-based OS that suppresses superuser privileges and exposes a narrow management interface consisting of a set of tailor-made brokers. Our evaluation shows that our modifications to Linux add negligible overhead to applications while preserving system manageability.

1 Introduction

Security threats related to system administrator ("admin") activity are receiving increasing attention, fueled by a series of events that highlighted the damage that such activities can inflict [6, 19, 22]. Traditionally, system maintenance requires superuser privileges for a range of operations. As a result, admins holding such privileges can put user data at risk through leakage, corruption, or loss. These hazards have raised concerns in many organizations [12, 13], and become even more relevant as companies [5] and government agencies [1] outsource IT management to third parties such as cloud providers. In the space of operating system design, in particular, these concerns have in part motivated research in "least-privilege" system designs that reduce the TCB size [16, 25, 37], offer more fine-grained protection [11], harden the TCB using formal verification [21], or use labeling to reason about and control information flow [33, 36].

A unifying goal underlying the existing body of work is to build *untrusted-admin* systems, i.e., systems that can be used by users who wish to store and

process sensitive data (either locally or "in the cloud") without requiring trust in the administrators of either their own systems or the cloud platform. The focus of this body of work has been on low-level kernel or hypervisor mechanisms, and little attention has been devoted to the higher-level challenges of building untrusted-admin systems that actually remain *administerable*. For example, in the influential Decentralized Information Flow Control (DIFC) model exemplified by the HiStar OS [36], building an untrusted-admin system requires not just the DIFC-enforcing kernel but also a set of user-level processes with *declassification* privileges, which users (data owners) must trust to handle their data appropriately during management activities that by nature must touch or affect this data. If a cloud system is to offer data backup services, for example, then the system must include some form of trusted daemon or declassifier that can read the user's data and forward it to the backup destination (perhaps after encryption). However, HiStar did not look into the problem of how to securely design these trusted daemons or declassifiers so as to cope with the range of management tasks performed by the admins.

The main challenge in finding a solution to this problem lies in a tension between security and manageability. In practice, operating systems require a wide variety of tasks to keep the system operational, some of which may touch or otherwise impact sensitive user data and processes, e.g., adding and removing software packages and drivers, loading kernel modules, applying security patches, managing user accounts, backing up and restoring user data, etc. Any of these "standard" administrative actions, if not handled carefully, could give an untrusted admin access to sensitive data either directly (e.g., a compromised backup daemon or declassifier) or indirectly (e.g., if an admin can "upgrade" a correct kernel module to an insecure version). Actually designing realistic and usable mechanisms and tools enabling administrators to *do their job* of managing OSes, in an untrusted-admin model, remains a largely unexplored challenge.

To address this challenge, we introduce and explore an untrusted-admin system design model that we call the *broker security model*, which we apply to the design of operating systems, but is a software system design model that can be applicable to a range of software systems. Our model is inspired by the central observation that users must in practice trust admins for *resource availability*, even if they do not wish to trust admins for *information security*. For example, a malicious or merely negligent admin can always "pull the plug," drop network connectivity, or fail to migrate data or virtual machines off of old hardware to be decommissioned. Such availability failures are typically obvious to users, however, and leave a clear "accountability trail"; a cloud provider will not survive as a business if it fails to maintain promised resource availability.

Thus, we aim to create a clean OS design separation between *resource availability* mechanisms, over which admins must have control in order to do their job, and *information security* mechanisms, over which admins *must not* have control. To meet this goal, our model only allows the admin to access and manipulate system objects in well-formed ways through a set of trusted programs called *brokers*. Brokers *never* concede the admin superuser privileges. Instead,

they only provide him with the specific functionality that is necessary to manage the system (e.g., create a user account) and control the resources (hence data availability) while ensuring that users gain control over the confidentiality and integrity of their data. To enforce this policy, the model defines three security invariants that the brokers must preserve; insofar as the these invariants are met, the system designer is free to specify the number and functionality of brokers that can better assist the admin in managing the system and to devise the most adequate mechanisms to enforce the model's security invariants.

To validate the broker security model, we present the design and implementation of a proof-of-concept OS called *Broker Umbrella for Linux-based OSes* (BrokULOS), which is based on a Debian Linux distribution enhanced with tailor-made broker extensions. One key design challenge is related to the fact that the management tasks in Linux are numerous, heterogeneous, and ill-defined. Since superuser privileges can no longer be granted to the admin, it is not clear which functions must be implemented to provide full OS manageability and whether these tasks can be performed without violating the model's security constraints. We address these questions by (1) characterizing the broker functionality based on a comprehensive survey of the fundamental tasks for maintaining a vanilla Debian distribution, and (2) specifying how exactly this functionality must be adapted in order to preserve the invariants of the broker model. We find that this functionality can be implemented by extending well-known Linux mechanisms and therefore show that the degree of protection proposed by the broker model is practical on commodity OSes, and does not require the use of niche research systems like HiStar (though these systems could also benefit from adopting our model).

In summary, our contributions are as follows. First, we characterize the important problem of enhancing the security of software systems in general (and OSes in particular) against administration threats while retaining system manageability. Second, we propose a principled way to approach this problem by introducing the broker security model. Third, we comprehensively study OS platforms and design BrokULOS, a system that demonstrates that enforcing this model on commodity OS platforms is possible with relatively few changes to existing Linux mechanisms. Finally, we evaluate our prototype, showing that BrokULOS preserves manageability, adds modest overhead to the management operations performed by the admin, and negligible overhead to the system.

2 Goals, Assumptions, and Threat Model

Our main goal is to devise a security model for enhancing the security of software systems against mismanagement threats. We focus, in particular, on administration roles that target the OS and require superuser privileges, e.g., installing applications, configuring devices, setting up security policies, creating user accounts, etc. We aim to find a sweet spot in the design space that strikes a balance between limiting the power of the admin and providing the functionality that is required for maintaining the system. We envision that the principles of our security model will be applicable to a range of software systems that currently

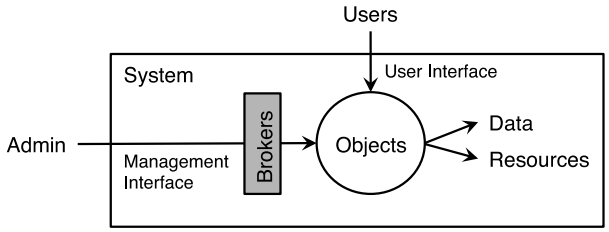

Fig. 1. Software system under the broker security model.

depend on granting superuser privileges in their specific domains (e.g., database servers or web applications). To demonstrate the feasibility of our model, our solution should not require deep changes to existing OSes and should mostly preserve compatibility with legacy applications.

Our model rests upon several assumptions. We consider that the implementation of the OS trusted computing base (TCB) is correct. Our focus is not minimizing the TCB size; such goal is complementary to our work and has been the focus of various other research projects [24, 25, 34, 37]. For this reason, our design is centered on a monolithic kernel with a large TCB. Nevertheless, we discuss in Section 8 a possible approach to reducing the TCB size by using an information flow kernel such as HiStar [36]. Additionally, we assume that the machine that hosts the system is physically secure, and that the system exposes a management interface that allows the admin to manage the system remotely. This situation is common in many organizations that host and process sensitive data, such as cloud providers [15].

We characterize our threat model. We assume that the admin has access to the machine through its management interface and uses the operations exposed by this interface to maintain the system and access user data (either on disk or memory). In a commodity OS, for example, this interface consists of all operations that require superuser privileges and therefore need to be performed from the root account or through a *sudo* gateway. In particular, if the management interface allows the admin to reboot the system, which is a necessary capability in the case of an OS, the admin can bypass the system security protections and have access to the persistent system state stored on disk. However, the admin cannot exploit vulnerabilities in the TCB code, for instance, to perform privilege escalation attacks, nor perform physical attacks on the machine. In addition, we do not consider side channel attacks.

3 Broker Security Model

The broker security model enhances the security of a software system by weakening the trust requirements relative to the system admin. In particular, it precludes the admin from compromising the confidentiality and integrity of user data and computations, while preserving manageability.

Figure 1 shows how the broker security model extends a software system. The base system (which follows a conventional system design) is modeled as a collection of objects, each of them containing data and holding a set of hardware

resources. In an OS, for example, these objects include files, processes, user accounts, etc. The system allows users and admin to access and manage objects through two interfaces—a user interface and a management interface. In the base system the management interface gives the admin superuser privileges, which allow him to fully control all system objects and therefore access user data without restrictions. Under the broker security model, however, the management interface no longer grants superuser privileges but only allows the admin to execute a set of trusted programs called *brokers*.

Brokers mediate the access to objects in a well-formed manner as to (1) provide the functionality that is necessary and sufficient to manage objects properly (e.g., create user accounts) and (2) let the admin retain control over resource availability while shifting control over user data confidentiality and integrity to users. To make sure that users retain control over their data security, brokers must maintain the following three security invariants:
1. Information security. A broker does not allow user data to be output or modified in ways that violate the confidentiality and integrity of that data. For example, allowing a debugger to be attached to a user process without the user being aware of or having authorized this operation violates this property.
2. Identity protection. A broker does not allow user identities and associated credentials to be hijacked or overridden. Otherwise, the admin could abuse this privilege to impersonate a user and access his data. For example, allowing the admin to change user passwords arbitrarily breaks this requirement.
3. System integrity. A broker ensures that the system can only transition between system states that preserve security invariants 1 and 2. For example, a broker cannot allow arbitrary kernel modules to be loaded because this feature could be exploited for privilege escalation: loading a malicious module could subvert brokers' security mechanisms.

This simple model can then be applied to enhance the security of software systems (and OSes in particular) by adopting a two-step methodology. First, one must specify the broker layer by identifying the *functionality* that the set of brokers need to offer while simultaneously obeying the three security invariants prescribed by the model. Second, one needs to devise the *mechanisms* that implement brokers' functionality and enforce the security invariants. We next apply these steps to an OS.

4 OS Broker Functionality

A natural way to enforce the broker security model in an OS is to start from a point that is secure by design yet overly restrictive, and then add carefully crafted brokers to regain manageability. In particular, a natural starting point is a design that forces the admin to operate from a regular user account, i.e., suppress the root account and prevent unrestricted execution of privileged commands through `sudo`. The challenge then becomes specifying and designing a set of brokers that (1) do not overlook functionality that is necessary for keeping the system administrable and yet (2) enforce the security invariants of the broker model.

To achieve this, we start with a thorough characterization of the set of commands that brokers should support by surveying the most fundamental management tasks performed by admins. The tools that support these tasks can then provide the baseline mechanisms needed to implement the brokers. Since these tools are likely to violate the broker model invariants, it is necessary to validate whether and how such violations take place so that we can enhance these tools to build brokers that satisfy the invariants.

Table 1 shows the list of tasks that we surveyed along with an indication of how the various tasks violate the three security invariants we listed previously. This list combines the results of two approaches. In a bottom-up approach, we studied a collection of packages and respective tools available in a basic Debian distribution, identified the functionality of each tool, and used our judgment to assess whether its functionality is fundamental for the admin. In a top-down approach, we studied the system administration literature and identified the high level tasks that an admin needs to perform. Overall, we manually inspected 902 executables included in 100 packages[4] and studied three different textbooks [14, 17, 35]. We then converged on a single (coarse-grained) task list, which we have examined with professional system administrators from the host institution of one of the authors to make sure it reasonably characterizes the management activity of a typical OS admin.

The tasks that violate the information security (IS) invariant mostly involve processes, files, and volumes and their primary goal is to manage resources and user data. For example, to learn about the memory utilization and open files by user processes, tools like `ps` and `lsof` reveal sensitive information that may be contained, e.g., in command line arguments of the process or in the names of user files. Similarly, tools for backing up and restoring user data (e.g., `tar` and `gzip`) would allow the admin to inspect and modify user data.

The tasks that breach the identity protection (IP) invariant are mostly related to user accounts and group management. User account operations include the ability to arbitrarily set and modify the identity and credentials of a user account (e.g., changing the password of an account using `passwd`). Group management enables adding and removing users from groups with tools like `useradd` and `usermod`. These capabilities would allow the admin to access files and processes owned by the user, in the first case, or shared within a group, in the second case.

The tasks that compromise the system integrity (SI) invariant are mostly related to software and system management. Typical OSes allow the admin to install arbitrary software, which can affect both the TCB (e.g., by upgrading the kernel, installing OS services, loading kernel modules) as well as shared applications. With this capability the admin could escalate his privileges to access user data by tampering with the TCB or by installing backdoors in shared applications. Admins can also set up devices to compromise the system integrity. For example, the ability to set the system time can be used to launch replay attacks.

[4] These packages were selected from a minimal Debian distribution according to two criteria: they contain the basic tools (package "Priority" is "Required" or "Important") and provide system administration support (package "Section" is "Admin").

Category	Management task	IS	IP	SI
Software	List, install, upgrade, and remove shared applications and libraries			×
	List, install, upgrade, and remove system services and kernel images			×
	Configure software and diagnose errors			×
	Apply security patches			×
	Manage local system documentation			×
Accounts	Create, modify, and delete user accounts		×	
	Disable user accounts temporarily			
	Modify account credentials		×	
	Force users to modify their credentials			
Groups	Create, modify and delete user groups		×	
Processes	Monitor and limit memory utilization by user processes	×		
	Check for runaway processes	×		
	Modify process execution priorities	×		
	Check for unattended login sessions	×		
Files	Perform backup and restore of user data	×		
	Set and view disk quotas			
	Check file space utilization	×		
	Remove temporary files (in `/tmp` and in `/lost+found`)	×		
	Re-distribute disk space in the filesystem	×		
	Mount and unmount filesystems			
	Check filesystem integrity and fight fragmentation	×		
	Check disk space	×		
	Create, modify, and format partitions			
System	Restart the system after panics, crashes, and power failures			
	Load, list, and unload kernel modules			×
	Start and stop services			×
	Automate and schedule system administration tasks with `cron`			
	Check and clear system log files	×		
	Configure and modify swap space			
	Configure `init` and runlevels			×
	Configure the network and check open connections	×		
	Setup system clock			×
	Setup and check the status of the printer			

Table 1. Management tasks grouped into categories: Tasks are grouped by category. For each task we indicate the security invariants they violate: information security (IS), identity protection (IP), and system integrity (SI).

Note that the purpose of Table 1 is not to enclose *all* management tasks. Instead, it comprises only the set of fundamental broker operations, which admins can then rely upon for more complex tasks. For example, for diagnosing resource misuse, admins can use various brokers, e.g., for checking runaway processes, unattended login sessions, and process memory utilization. In fact, it is typical to use helper tools to identity the source of such problems. Another example, for recovering from system bugs, admins can use brokers for securely installing software and backing up / restoring user data. Indeed, rather than fixing compromised systems, the common practice for system recovery is to make clean-slate software reinstalls and restore user data from backups; this method guarantees that the system state is again known and trustworthy.

Ideally, the table should list all the tasks that are necessary and sufficient to meet all needs of OS admins. In spite of our best efforts and positive feedback

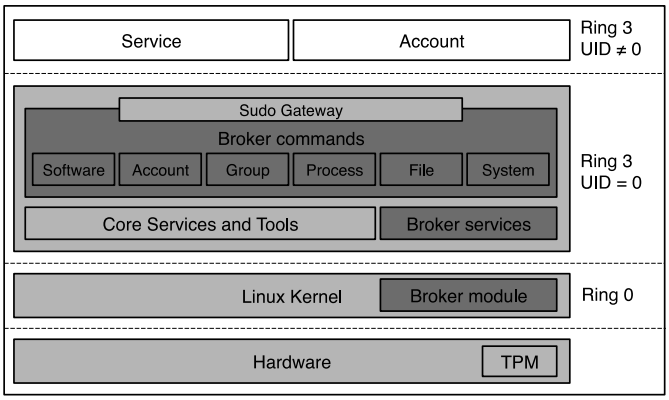

Fig. 2. Broker-enhanced OS architecture.

from expert system administrators, however, this table is not necessarily complete and may need to be adapted by adding, modifying, or removing entries depending on the concrete OS, deployment environment, and admin needs.

Now that we have characterized the functionality that should be offered by the broker layer, we present an OS design that implements it.

5 Broker-enhanced OS Design

We start with an overview of the OS architecture that we propose and then describe how each security invariant is enforced by the brokers.

5.1 Architecture

Figure 2 illustrates the internals of a broker-enhanced OS. Since it is not our primary goal to minimize the size of the TCB, we simply extend a vanilla Debian Linux distribution with a set of components that implement the broker extensions for the system. These components consist of broker commands, dedicated services, and an LSM kernel module.

In contrast to the vanilla Debian distribution, there is no superuser account (root) nor any other way that the admin can obtain superuser privileges. Instead, both users and the admin run their processes in protection domains with UID > 0. UID 0 is then reserved for the components that need to run in privileged mode such as OS services (e.g., `init`, `sshd`), and broker commands. The space of unprivileged domains (UID > 0) is split into two parts: UIDs $\leq u_t$, which are reserved for services that do not need to run in privileged mode, and UIDs $> u_t$, which are reserved for user accounts (where u_t is a configurable threshold).

Brokers consist of a well-defined set of trusted programs that run in privileged mode (UID $= 0$). Table 2 shows examples of the most representative brokers, grouped into categories according to their semantics. To allow for invoking brokers from a non-privileged account, we rely on the well known `sudo` gateway, which also authorizes broker execution based on the role—admin or

user—associated with each account. To bootstrap the creation of admin accounts, the admin role is assigned to the first account to be created; the admin can then define the role of the subsequent user accounts.

Next, we describe in more detail the mechanisms introduced by the broker extensions that provide support for the management tasks in Table 1 while preserving the security invariants required by the model. We structure this presentation according to the invariants that are to be preserved.

5.2 Enforcing the Information Security Invariant

The information security invariant stipulates that the admin cannot access user data through the system management interface. This is the model's most fundamental requirement because otherwise user data confidentiality and integrity could be directly violated. To meet this requirement, the protection domains of the admin and users should be perfectly isolated from each other. However, this can be challenging when user domains must be crossed over, particularly for resource management and data management tasks. We discuss these in turn.

Managing Account Resources The admin must be able to control the resources associated with a user account (e.g., set user quotas for CPU and memory). This control, however, requires permission to access the resources allocated to user data. Without the proper protections, however, such access could allow the admin to access user data, thereby compromising its confidentiality and integrity. To enforce a clean separation between resources and data, we propose taking the following steps.

The first step is to conservatively isolate the protection domains of admin and users. To start, we can use the UID-based protection domains to prevent direct access to user files and processes that are not explicitly shared by the users. However, it is also necessary to prevent information leakage through the `/proc` filesystem. The Linux kernel exposes extensive information relative to user processes in a collection of files located under `/proc/`*`PID`*, where *`PID`* is the process number. The kernel generates the content of these files on the fly whenever they are opened and sets the permissions of many of them to publicly readable. However, making some of these files public violates the information security invariant (e.g., files `stat` or `cmdline` expose many details about the memory usage or the command line of processes, respectively). To prevent access to this information with minimal kernel changes, we simply override the file permissions to make them private to the process owner and accessible to the system brokers. We preserve kernel compatibility by adding these changes in an LSM module.

To enable the admin to manage account resources, the second step is to provide a set of specific brokers for process and file management. These brokers, however, only let the admin "see" an account as a bundle of CPU, memory, and storage resources whose utilization he can observe, restrict (by setting quotas), and deallocate *as a whole*. For example, brokers for process management only output aggregate information of resource utilization and always operate on all processes of an account (e.g., by applying `kill` and `renice` to all processes).

Category	Examples of representative brokers
Packages	list packages (`pkg-list`), get package (`pkg-get`), install package (`pkg-install`), upgrade package (`pkg-upgrade`), remove package (`pkg-remove`), flush package cache (`pkg-flush`)
Accounts	create account (`acc-create`), disable account (`acc-disable`), enable account (`acc-enable`), force password reset (`acc-force`), reset password (`acc-passwd`), delete account (`acc-delete`), load user policy (`acc-polload`)
Groups	create group (`grp-create`), list groups (`grp-list`), delete group (`grp-delete`), add member (`grp-addmem`), list members (`grp-lstmem`), remove member (`grp-remmem`)
Processes	list resource utilization (`ps-list`), kill account processes (`ps-kill`), set account process priority (`ps-renice`)
Files	backup account files (`fls-backup`), restore account files (`fls-restore`), list storage usage (`fls-du`), move account (`fls-move`), clean temp (`fls-cltmp`)
System	insert module (`mod-insert`), remove module (`mod-remove`), list services (`svc-list`), start service (`svc-start`), stop service (`svc-stop`), reboot (`sys-reboot`), setup system clock (`dev-clock`), setup network card (`dev-net`)

Table 2. List of representative brokers grouped into categories: Describes each broker's functionality and command name (in parenthesis).

Brokers for file management follow the same approach. As another example, monitoring the storage consumed by a user and moving user files to another volume only reveals aggregate disk utilization and displaces all files located in users' home directories or in user-approved subdirectories, respectively.

Exporting Account Data The aforementioned techniques allow for resource management without user data access. However, in certain operations like backing up and restoring user data the admin needs to export user data from the user account's protection domain, where the data is secured, to another machine. To support these operations while preserving information security, the system encrypts the data and appends integrity checks before the data leaves the protection domain. However, we need to ensure that, when restoring the data, the backed up data can only be decrypted (1) on machines booting an untampered version of BrokULOS and (2) by the original owner of the data. To guarantee this property, the user data is encrypted and decrypted with a *seal key*. The seal key is a unique cryptographic key that the system associates with each newly created account. To enforce requirement (1), we take advantage of TPM primitives, which allow us to encrypt (seal) the seal key such that it can only be decrypted (unsealed) if the machine boots a correct BrokULOS binary. If the booted system is correct, the system then ensures that the seal key is only accessible to the owner's account, thereby ensuring requirement (2). To support recovering data on a different machine, e.g., because the original one was decomissioned, sealing could be extended to allow for unsealing to take place on any machine with a similar configuration. This extension could be done by coupling BrokULOS with Excalibur [32], a trusted computing system that enforces access control policies for multi-node environments.

5.3 Enforcing the Identity Protection Invariant

With the protection mechanisms for the enforcement of information security in place, the admin no longer has direct access to user data. Nevertheless, these

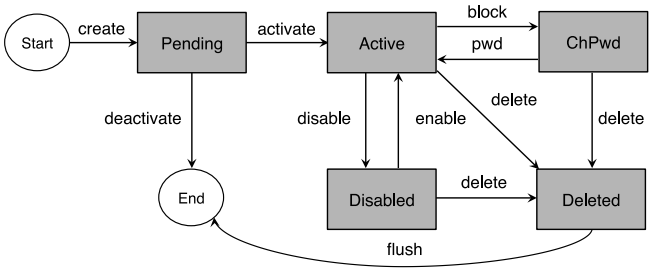

Fig. 3. State transitions between account states: The user must explicitly accept that the account is valid before it can be used. In the active state, the admin can temporarily disable the account or force the user to change authentication credentials. The resources of a deleted account can be released at a later point in time.

protections could be circumvented if the identity protection invariant is not assured. This invariant requires that the admin cannot control user credentials and identities, otherwise he could impersonate users and access their data directly. Thus, ideally, users should be able to control their own identities without hindering the admin's ability to control resources. In practice, however, shifting control to users entails some loss of management flexibility for the admin. Therefore, we need to design brokers for managing accounts and groups that provide reasonable manageability without sacrificing the identity protection invariant, as we describe next.

Managing User Accounts In managing user accounts, we enforce the identity protection invariant by offering a set of brokers for regulating an account's life cycle such that user login credentials are strictly controlled by the user.

The basic life cycle of a user account is shown in Figure 3. An account is created by the admin; he specifies the initial configuration of the account (e.g., user name, home directory) and an initial login credential, which is only going to be used once. The first time the user logs in with the initial login credential, he must ensure that he has exclusive access to the account by claiming it. This process involves running a secure protocol which serves two purposes. First, it provides a report describing the initial account's configuration and state. If the account has been set up with initialization scripts or if somebody has logged into it before, the user will be able to detect these irregularities and abort the operation. If, however, the report shows no problems, the user can set up his authentication credentials (e.g., by uploading the user's public key) without admin interference. This process will disable the initial login credential and lock the user name associated with the account. From this point onwards, only the user can login to his account and he has full control over its content, but not its resources. The admin can still adjust the resources associated with the account, disable user login temporarily (e.g., in the case of a misbehaving user), force a user to change credentials, and, whenever necessary, delete the account.

Changing credentials is done by users themselves using the credentials they have uploaded to the system. To address the concern that losing user credentials would prevent a user from ever logging in, our system supports two override

mechanisms. One is to rely on a trusted third party, either a single entity or a quorum, to reset the user credentials. Another is to increase redundancy by registering multiple credentials and using various authentication mechanisms (e.g., public key, password, passphrase). Although this approach does not eliminate the problem entirely, it reduces the likelihood of permanent loss of access.

Managing Group Membership Aside from allowing users to control their own identities and credentials, user groups' members need to be properly authenticated. Otherwise, the admin could gain access to group-shared data by creating fake identities and registering them as legitimate group members. To enforce the identity protection invariant when managing groups, the BrokULOS admin is still allowed to create and delete user groups, but adding and removing members is delegated to users themselves. The approach we use for delegation is to designate a (per-group) *group leader* that makes group membership authorization decisions. The group leader must validate users' identities before adding them to a group. Since relying on user names chosen by the admin is insecure for authentication, the group leader must check users' credentials (e.g., a certificate of the user's public key).

5.4 Enforcing the System Integrity Invariant

The mechanisms we have introduced thus far can effectively enforce both the information security and identity protection invariants. However, if the admin can compromise these mechanisms, these assurances can no longer be guaranteed. Thus we next propose a mechanism for enforcing the system integrity invariant, taking into account two aspects of the problem: managing TCB components and shared applications.

Managing TCB Components Managing TCB components involves installing, upgrading, configuring and removing software components that run in privileged domains—either in the kernel space (i.e., the kernel itself or kernel modules) or in the user space with UID 0 (e.g., services, system libraries, system tools, and brokers)—and configuring devices (e.g., setting up the network and the system timer). To enforce the integrity of the TCB, all these operations must be validated, and this is done using special-purpose brokers.

In particular, for installing TCB components, brokers only authorize this operation if the new TCB component is "trusted". Several definitions of trust could be used, for example, in an ideal world, the system would automatically verify if the implementation is correct. BrokULOS uses a simple model where a TCB component is trusted if its compliance with the broker security model is endorsed by one or multiple third parties that are mutually trusted by both the admin and users, referred to as Mutually Trusted Signers or MTSes. (Users' consent is necessary otherwise a misbehaved admin could use this mechanism to modify the TCB.) To enforce this consent, admins set up the initial MTS certificates in the system and users must approve or reject them whenever they claim their accounts. MTS certificates can be changed over time—e.g., when renovating or

revoking them, or when adding new MTSes—by either establishing a chain of trust that only accepts new MTS certificates signed by a preexisting MTS, or by polling all users before accepting a new MTS certificate. The MTS role can be performed by any entity mutually approved by admin and users (e.g., certification organizations, software development companies, specific administration roles within the organization, or open source communities).

Regarding device configuration, we again only accept configurations that are vouched for by an MTS. The notion of what is expected from a trusted configuration is device-specific. Therefore device-specific brokers are expected to perform the appropriate validations. A particularly interesting case is the system clock, where the system time should not be set arbitrarily. Therefore, we restrict time updates to trusted NTP servers sent over secure channels. This is done by requiring the NTP configuration file (which identifies addresses and credentials of the NTP servers) to be signed by an MTS. Given the large number of devices, we did not design brokers for all of them, but new devices could easily be accommodated by incorporating appropriate brokers.

In addition to enforcing TCB integrity, it is necessary to assure users of its enforcement. This is because the admin can circumvent the TCB protection mechanisms by rebooting the machine and tampering with the TCB binaries on disk. We offer these guarantees by means of a remote attestation protocol, which users run when they claim their accounts. Our protocol is based on a standard attestation protocol [30], which transmits the boot time measurements (hash) of the TCB components signed by the TPM. We then extend it to include the MTS identities as well as the user account report (see Section 5.3). Thus, when users claim their accounts they can validate the hashes of the TCB binaries and the MTS identities, thus assessing the integrity of the TCB.

Managing Shared Applications Finally, another type of software that must be trusted to correctly manipulate user data are shared applications (e.g., MySQL). To give users the flexibility of choosing which applications they trust, we let them define *user policies* that express their restrictions. The policy language expresses a list of rules, each of them consisting of comparisons among four attributes we currently support: package maintainer, package name, package version, and filename.

To enforce these policies, we developed a special purpose LSM kernel module. The LSM module overrides the standard DAC permissions and enforces the user policy at runtime: whenever the user runs an external program, the LSM module intercepts this operation, evaluates the policy, and aborts the execution if the policy evaluation fails. To evaluate each policy rule, the LSM module checks the attribute conditions specified in the policy against a set of extended filesystem attributes featuring the executable. The filesystem attributes are attached by the broker layer whenever the executable's package is installed. The broker responsible for installing the packages obtains the attributes for each program from a manifest contained in the program's package. Users load their policies into the LSM module once they claim their accounts.

6 Implementation

Our BrokULOS prototype is based on the Debian GNU/Linux 6.0 ("Squeeze") distribution running Linux 2.6.39.3. Our implementation effort includes the broker layer, which we implemented in about 4,400 lines of Python code, and the LSM kernel model, coded in less than 1,000 lines of C code. For convenience, brokers take advantage of basic tools such as `dpkg`, `gpg`, and `useradd` to perform the low level changes to the system. These tools are included in the core packages of BrokULOS, which comprises 77 packages, out of a total of 266 packages. This package configuration is based on Debian's minimal setup, which is then extended with BrokULOS's functionality.

The LSM module implements the protection mechanisms for overriding the DAC permissions of the `/proc` files and evaluating user policies. To implement this functionality, it places handlers in two LSM hooks (bprm_check_security and inode_permission). The LSM module provides an interface via VFS under the mount point `/brokulos` for loading the user policies into the module.

Our current prototype uses TPMs to support remote attestation and secure storage. We use TrustedGRUB [3] to measure the integrity of the files of core packages and extend the PCR registers with these measurements accordingly. Then, we use the TPM's `quote` primitive to generate and sign an attestation report when requested by the users. This procedure requires setting up an AIK key so that the TPM can sign the report. The implementation of secure storage has some limitations: we keep the entire system on an encrypted partition using LVM, but, as of now, we have not modified LVM so that the encryption keys are protected using the sealing primitives of the TPM. This technique, however, poses no particular challenges and is already used in Windows by BitLocker [26].

7 Evaluation

We now evaluate the security, manageability, and compatibility of BrokULOS, and experimentally gauge its performance overheads.

7.1 Security

BrokULOS improves security in three main aspects. First, it significantly reduces the management interface exposed to the admin. Unlike a commodity Linux distribution where the admin is endowed with superuser privileges, in BrokULOS the admin can only perform the privileged operations exposed through the broker layer. The broker layer makes the management interface explicit, and narrows it to a relatively small numbers of trusted programs. Thus, provided these programs are correctly implemented, the admin cannot acquire privileges not contemplated in the broker model.

Second, BrokULOS explicitly restricts the software that can run in a privileged domain, i.e., that belongs to the TCB. In a commodity Linux distribution, because the admin can install arbitrary software in the privileged protection domain, it is not possible to foresee which security properties are guaranteed by the

system. In BrokULOS, however, only the software that is signed by an MTS can run in the privileged domain. Thus, provided that MTSes are trustworthy, the system enforces the well-defined security invariants of the broker model.

Finally, BrokULOS allows users to specify the software they trust to process their data. BrokULOS conservatively prohibits the execution of all shared programs (i.e., not owned by the user) and allows the user to open exceptions based on a user policy. This mechanism prevents the user from accidentally running applications that could compromise the security of his data.

An orthogonal aspect of the system security is shrinking the TCB size to reduce the likelihood of code vulnerabilities. As we mentioned, this aspect was not the emphasis of our work and we therefore see it as being complementary and a follow up to BrokULOS. Nevertheless, we note that while brokers add code to the TCB, it is only a small additional fraction of much simpler code when compared to the OS kernel. Furthermore, we expect to make broker programs trustable by releasing their source code.

7.2 Manageability

The ideal way to evaluate the system manageability would be through the practical experience of deploying and managing the system in a real setting. Not having access to such a deployment, our methodology is to validate the whether BrokULOS provides adequate broker coverage to accommodate all the management tasks we have surveyed (see Table 1).

Our current prototype provides a set of 41 brokers spanning multiple task categories. In some cases there is a one-to-one correspondence between the task and a particular broker (e.g., backing up data is supported by `file-backup`), whereas in others a single broker serves multiple tasks (e.g., `ps-list` lists both the CPU and memory allocated to an account). Overall, BrokULOS currently covers the most crucial set of management tasks. We provide only limited support for tasks related to devices (e.g., managing the printer) and filesystems (e.g., format partitions and fight fragmentation). Overall, out of the 33 coarse-grained tasks of the table, our system fully supports 29. Although devising brokers to support the remaining tasks constitutes a challenge when compared to the brokers we have built so far, the high fraction of management tasks covered by the existing brokers shows that our system provides extensive management support.

7.3 Compatibility

Overall, BrokULOS preserves compatibility with existing Linux mechanisms and applications. Our solution requires no modifications to the Linux kernel besides plugging in a kernel module to the standard LSM interface. The system leaves ABI / APIs unchanged, thereby preserving application compatibility. However, some popular administration tools are disabled, since they violate the broker model. This is the case, for example, `lsof`, which prints out a list of every file that is in use in the system. As a result, the admin may have to adapt and possibly change his scripts to use BrokULOS's brokers.

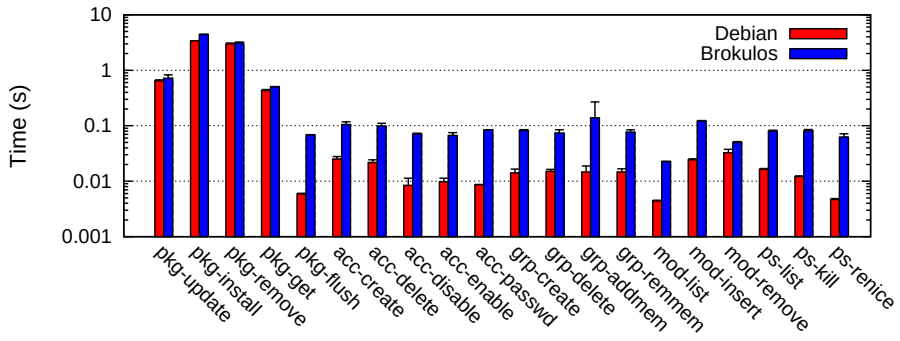

Fig. 4. Performance of brokers when executed by the admin: Covers representative brokers relative to package, account, group, module, and process management. The brokers for installing, getting, and removing packages use the `hello` package, which suffices for measuring the broker overhead for any package.

7.4 Performance

To evaluate the performance of our prototype, we focused on the places where BrokULOS introduces overheads to the vanilla Debian distribution: the broker layer, which affects management operations, and the LSM module, which impacts the execution latency of all programs in the system. (Recall that the LSM handler code runs every time the `exec` system call is executed.)

Our evaluation methodology is as follows. To study the broker layer overhead we use microbenchmarks. For each broker, we measure its execution time, measure the execution time of a vanilla Debian operation whose functionality is comparable to the broker's (e.g., user account creation), and then compare both values to analyze the performance penalty incurred by BrokULOS's management tasks. For each experiment, we run 10 trials and report the mean time and standard deviation. To study the overhead of the LSM module we measure the impact of policy evaluation on the execution time of a large task, namely compiling the Linux kernel 2.6.39.3. We measure the overall execution time with and without policy evaluation, using a policy with 266 rules, where each of them tests a package installed in the system. We use an Intel Xeon machine with a 2.83GHz 8-core CPU, and 1.6GB of RAM.

Figure 4 plots the results of the broker layer evaluation. It shows only the subset of system brokers that (1) require sanitization of standard admin tools to enforce compliance with the broker model (e.g., reseting the network card is not shown), and (2) have a direct correspondence with a vanilla Debian operation (e.g., the backup broker is not shown). There is a significant disparity in the performance overhead among brokers. Brokers whose Debian counterpart execute in the order of 10ms undergo a performance penalty of around one order of magnitude. For execution times above the 0.1s threshold, however, the performance between the two cases is comparable. The high overhead of short-lived brokers is partly due to the extra functionality, but mostly due to being implemented in Python, whereas their Debian counterparts are implemented more efficiently in C. If we consider, e.g., the `ps-renice` broker, which sets the same

priority to all the processes of a user, and its counterpart, which corresponds to the command `renice -u`, the 10-fold increase is simply due to Python overhead. Since the broker functionality is not significantly more complex than that of pre-existing tools', we believe that implementing brokers from scratch and in C should produce comparable performance to the Debian distribution.

Our LSM module study shows that policy evaluation is efficient. The overall execution times of the kernel compilation in Debian and in BrokULOS show no differences, which means that the LSM module adds negligible overhead to long running tasks. These results are expected since the LSM module handlers perform very little work and only when a program is executed.

8 Discussion

In this section we discuss several issues regarding possible design extensions and the deployment of the system.

Shrinking the TCB size. Several directions could be taken to reduce BrokULOS's TCB size. One direction is to leverage existing sandboxing mechanisms for Linux such as UserFS [20] in order to run some of the trusted programs (e.g., privileged services) in an unprivileged environment. Thus, exploiting one of these services would not compromise the entire system. To avoid depending on the correctness of the large Linux kernel, a second direction is to explore designs based on microkernels [21] or on DIFC kernels [23, 36]. The important thing to note is that the broker security model is also applicable in these settings, with the added advantage that brokers can set fine-grained policies; e.g., the `ps-list` broker can be constrained to only be able to read the `/proc` files. Thus, in the event of an exploit, the attacker could only leak information from those files and nothing else, thereby improving security.

Extension to medium- and large-scale deployments. In real deployments, a machine rarely operates autonomously; it may rely on networked services for storing data (e.g., NFS), authentication (e.g., LDAP), or upgrading software (e.g., package repositories), for example. In cloud computing or grid platforms, each machine is itself a constituent of a larger distributed system. Although in this paper we have focused on securing a single machine, we believe that the same principles can be applied to a distributed setting by propagating trust across components using secure channels and remote attestation mechanisms. However, we have not yet explored these extensions.

9 Related work

We organize related work into security models, systems that restrict admin privileges, and security mechanisms for Linux.

Security models. Bell-LaPadula [8] and Biba [9] are well known information flow security models for multilevel security that can express confidentiality and integrity policies, respectively. These and other IFC models [28], however, focus on how information flows in a system and have not looked at expressing the range of management operations required by admins (e.g., for managing software),

which is the focus of our work. Clark-Wilson (CW) [10] is an informal security model concerning data integrity, which aims to prevent users from manipulating data objects arbitrarily. Our broker model shares similarities with CW in that CW also relies on trusted programs to streamline the way data objects can change. In contrast, we focus not on users' access control but on admins', and we go beyond CW in prescribing concrete invariants that trusted programs must adhere to in order to secure the system's management interface.

Systems that restrict admin privileges. Despite apparent similarities with some of our design choices, current commodity OSes rely on a fully trusted admin. In particular, although Ubuntu [4] does not have a root account, the admin can still acquire superuser privileges and perform arbitrary operations through a trusted program. The Plan9 [31] distributed system was the first OS without superuser. Plan9 comprises multiple nodes, each of which is managed independently by a node's owner. Although there is no system-wide superuser, the owner of each node can control not only the node resources, but also compromise the security of the user data located on the node. More recently, HiStar [36] showed that the separation between resource management and data management is possible using DIFC. However, HiStar only provides the DIFC foundations for data protection and does not consider the high-level manageability issues addressed in BrokULOS. Similarly, trusted computing systems [25, 33] have focused on securing user data and computations from the admin by using confinement [25] and labeling [33] techniques, but without specific concerns for preserving manageability. In the hypervisor world, the work by Murray et al. [27] and more recently CloudVisor [37] allow for management of VMs without admin interference, but address different challenges than BrokULOS's, which targets OSes rather than virtualized platforms.

Security mechanisms for Linux. Many mechanisms have been specifically designed to improve Linux security. A large body of these mechanisms aim to confine untrusted code to some kind of sandboxing environment, e.g., `chroot`, Jails [18], Linux containers [2], and UserFS [20]. Other mechanisms such as SELinux [29] and AppArmor [7] provide some specific support for MAC in Linux. Each of these mechanisms cannot per se address the manageability issues that constitute the focus of our work. Nevertheless, some of these proposals share similarities with BrokULOS's user policies. SELinux also allows defining policies based on specific programs, but it differs from BrokULOS in that SELinux policies are defined by the admin, whereas BrokULOS's policies are defined by the users. AppArmor allows attaching policies to programs based on file paths, which BrokULOS also supports. However, in AppArmor, if a program has no policy associated with it, then it is by default not confined. Thus, contrary to BrokULOS, it cannot protect users from accidentally executing malicious programs not covered by the policies. Note, however, that BrokULOS's key contribution is not so much in proposing fundamentally new mechanisms, but in showing that, by putting together and adapting well known Linux mechanisms, enhancing Linux according to the broker model is possible, adds little impact to performance, and provides good manageability.

10 Conclusion

This paper introduced the broker security model, a general security model aimed at protecting the confidentiality and integrity of user data from system administration errors. By only trusting admins for resource availability and not for information security, this model improves data protection with little impact on system manageability. It achieves this property by relying on a layer of *brokers*—trusted programs that mediate access to system objects. We showed that this model is practical for OSes by implementing and evaluating BrokULOS, our proof-of-concept broker-compliant OS. The broker model lays out important principles in the design of untrusted-admin systems. We envision applying it to other software systems (e.g., databases and web applications) and improving the mechanisms necessary to enforce this model (e.g., by reducing the TCB size).

Acknowledgements. We would like to thank Carina Schmitt and Jörg Herrmann for sharing with us their experience as professional system admins. We are also grateful to the anonymous reviewers for their feedback. This work was partly supported by the National Science Foundation under grant CNS-1149936. The research of Rodrigo Rodrigues is supported by an ERC starting grant.

References

1. Federal Government's Cloud Plans: A $20 Billion Shift, http://www.cio.com/article/671013/Federal_Government_s_Cloud_Plans_A_20_Billion_Shift
2. Lxc Linux Containers, http://lxc.sourceforge.net
3. Trusted GRUB, http://trousers.sourceforge.net/grub.html
4. Ubuntu, http://www.ubuntu.com/
5. Verizon to Put Medical Records in the Cloud, http://www.networkcomputing.com/cloud-computing/229501444
6. Insecurity of Privileged Users: Global Survey of IT Practitioners. Tech. rep., Ponem Institute and HP (2011), http://h30507.www3.hp.com/hpblogs/attachments/hpblogs/666/62/1/HP%20Privileged%20User%20Study%20FINAL%20December%202011.pdf
7. AppArmor, http://www.novell.com/linux/security/apparmor
8. Bell, E.D., La Padula, J.L.: Secure computer system: Unified exposition and Multics interpretation. Tech. rep., MITRE Corp. (1976)
9. Biba, K.J.: Integrity considerations for secure computer systems. Tech. rep., MITRE Corp. (1977)
10. Clark, D.D., Wilson, D.R.: A Comparison of Commercial and Military Computer Security Policies. In: IEEE Symposium on Security and Privacy (1987)
11. Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., Warfield, A.: Breaking up is hard to do: security and functionality in a commodity hypervisor. In: SOSP (2011)
12. ENISA: Cloud Computing - SME Survey (2009), http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-sme-survey/
13. ENISA: Cloud Computing Risk Assessment (2009), http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment
14. GBdirect: Linux System Administration (2004), http://training.gbdirect.co.uk

15. Hamilton, J.: An Architecture for Modular Data Centers. In: CIDR (2007)
16. Härtig, H., Hohmuth, M., Feske, N., Helmuth, C., Lackorzynski, A., Mehnert, F., Peter, M.: The Nizza Secure-system Architecture. CollaborateCom (2005)
17. Josep Esteve and Remo Boldrito: GNU/Linux Advanced Administration (2007)
18. Kamp, P., Watson, R.N.M.: Jails: Confining the omnipotent root. In: SANE'00 (2000)
19. Keeney, M.: Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors. Tech. rep., U.S. Secret Service and CMU (2005), http://www.secretservice.gov/ntac/its_report_050516.pdf
20. Kim, T., Zeldovich, N.: Making Linux Protection Mechanisms Egalitarian with UserFS. In: USENIX Security Symposium'10 (2010)
21. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP (2009)
22. Kowalski, E.: Insider Threat Study: Illicit Cyber Activity in the Information Technology and Telecommunications Sector. Tech. rep., U.S. Secret Service and CMU (2008), http://www.secretservice.gov/ntac/final_it_sector_2008_0109.pdf
23. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information Flow Control for Standard OS Abstractions. In: SOSP (2007)
24. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V.D., Perrig, A.: TrustVisor: Efficient TCB Reduction and Attestation. In: IEEE Symposium on Security and Privacy (2010)
25. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: EuroSys (2008)
26. Microsoft: BitLocker Drive Encryption, http://www.microsoft.com/whdc/system/platform/hwsecurity/default.mspx
27. Murray, D.G., Milos, G., Hand, S.: Improving Xen Security Through Disaggregation. In: VEE (2008)
28. Myers, A.C., Liskov, B.: A Decentralized Model for Information Flow Control. In: SOSP (1997)
29. NSA: Security-Enhanced Linux (SELinux) (2001), http://www.nsa.gov/selinux
30. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Commodity Computers. In: IEEE Symposium on Security and Privacy (2010)
31. Russ Cox and Eric Grosse and Rob Pike and Dave Presotto and Sean Quinlan: Security in Plan 9. In: USENIX Security Symposium'02 (2002)
32. Santos, N., Rodrigues, R., Gummadi, K.P., Saroiu, S.: Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In: USENIX Security (2012)
33. Sirer, E.G., de Bruijn, W., Reynold, P., Shieh, A., Walsh, K., Williams, D., Schneider, F.B.: Logical Attestation: An Authorization Architecture for Trustworthy Computing. In: SOSP (2011)
34. Steinberg, U., Kauer, B.: NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In: Eurosys (2010)
35. Wirzenius, L., Oja, J., Stafford, S., Weeks, A.: The Linux System Administrator's Guide (1993-2004), http://tldp.org/LDP/sag
36. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making Information Flow Explicit in HiStar. In: OSDI (2006)
37. Zhang, F., Chen, J., Chen, H., Zang, B.: CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In: SOSP (2011)