Practical Techniques to Obviate Setuid-to-Root Binaries

Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter
Stony Brook University
{*bpjain, chitsai, jijjohn, porter*}*@cs.stonybrook.edu*

Abstract

Trusted, setuid-to-root binaries have been a substantial, long-lived source of privilege escalation vulnerabilities on Unix systems. Prior work on limiting privilege escalation has only considered privilege from the perspective of the administrator, neglecting the perspective of regular users—the primary reason for having setuid-to-root binaries.

The paper presents a study of the current state of setuid-to-root binaries on Linux, focusing on the 28 most commonly deployed setuid binaries in the Debian and Ubuntu distributions. This study reveals several points where Linux kernel policies and abstractions are a poor fit for the policies desired by the administrator, and root privilege is used to create point solutions. The majority of these point solutions address 8 system calls that require administrator privilege, but also export functionality required by unprivileged users.

This paper demonstrates how least privilege can be achieved on modern systems for non-administrator users. We identify the policies currently encoded in setuid-to-root binaries, and present a framework for expressing and enforcing these policy categories in the kernel. Our prototype, called Protego, deprivileges over 10,000 lines of code by changing only 715 lines of Linux kernel code. Protego also adds additional utilities to keep the kernel policy synchronized with legacy, policy-relevant configuration files, such as `/etc/sudoers`. Although some previously-privileged binaries may require changes, Protego provides users with the same functionality as Linux and introduces acceptable performance overheads. For instance, a Linux kernel compile incurs less than 2% overhead on Protego.

Net lines of code de-privileged.	12,717
Percentage of deployed Ubuntu and Debian systems that can eliminate the setuid bit.	89.5%
Historical exploits that would be unprivileged on Protego.	40/40
Performance overheads	$\leq$7.4%
System calls changed	8

Table 1. Summary of results.

1. Introduction

Unprivileged users of modern Unix systems access safe subsets of otherwise privileged system functionality through trusted, setuid-to-root binaries. For instance, the `mount` utility executes with administrative privilege, allowing unprivileged users to mount a CD-ROM or USB Flash device without involving a human administrator. Because the `mount` binary must issue the `mount()` system call, which requires root privilege (or the `CAP_SYS_ADMIN` capability), the `mount` binary is inadvertently empowered to issue many more privileged system calls. For instance, if an attacker can exploit an input parsing bug in `mount`, she might be able to change the root user's password, install a rootkit, or replace the contents of the `/etc` directory. Chen et al. [15] show that many privilege escalation attacks go through setuid-to-root binaries, even on SELinux [32] or AppArmor [6]. This attack surface is ubiquitous; e.g., 99.99% of surveyed systems install `mount` (§3.3).

The problem with setuid-to-root binaries is that they violate the Least Privilege Principle (LPP) [38], and create opportunities for privilege escalation attacks. As a result, most major Linux distributions have ongoing, but incomplete efforts to prune unnecessary setuid-to-root binaries [21, 45] (§3.1). Although these efforts have made substantial progress in reducing the number of setuid-to-root binaries, they leave a small core of "unavoidable" trusted binaries that continue to violate least privilege.

Previous research efforts [13, 23, 25, 29, 46, 47], as well as hardened Linux configurations, such as SELinux and AppArmor, have only considered least privilege on these utilities from the perspective of the administrator, not the untrusted user. In the case of mount utilities, systems like AppArmor attempt to limit the effects of a compromised `mount` to arbitrarily changing the file system tree. When the administrator executes `mount` with least privilege, she can only corrupt the file system tree, not change passwords (directly) or configure the network device. In contrast, least privilege for an unprivileged user should restrict that user to only mounting white-listed devices and directories (e.g., `/cdrom`); even on AppArmor, a bug in `mount` could allow an unprivileged user to make arbitrary changes to the file system tree. SELinux further restricts `mount` to specific users and mountpoints, but still trusts `mount` to correctly map devices and options to these mountpoints. In this ex-

EuroSys 2014, April 13 - 16 2014, Amsterdam, Netherlands

ACM 978-1-4503-2704-6/14/04…$15.00.
http://dx.doi.org/10.1145/2592798.2592811

ample, least privilege on these utilities can only be enforced by the OS kernel, and policies must be expressed not just in terms of the user requesting a system call, but also in terms of the objects of the requested call.

This paper presents a simple, efficient framework for migrating policies from setuid-to-root binaries into the kernel, obviating the need for these privileged binaries in nearly all situations. We study the 28 most commonly installed binaries on Debian and Ubuntu Linux, which account for all setuid-to-root binaries on roughly 89.5% of systems surveyed (§3.3).

One essential insight from the study is that only eight system calls and a few other system interfaces underlie the vast majority of root privilege requirements. These system calls export functionality that is required by unprivileged users, yet require administrative privilege. For these system calls, this paper proposes enforcing more expressive policies that more closely match the policies encoded in setuid binaries and configured by administrators. We present a prototype, called **Protego**, which extends the AppArmor [6] Linux Security Module (LSM) [48]. Protego changes only 715 lines of Linux kernel code, and adds additional trusted utilities to keep the kernel policy synchronized with legacy, policy-relevant configuration files, such as `/etc/sudoers`. In addition to these 8 system calls, a few additional system abstractions must be adjusted to remove privilege. Although Protego extends AppArmor, any security-hardened Linux variant could adopt these techniques.

An underlying motivation for setuid-to-root binaries is flexibility. When the kernel adopts a new abstraction, system developers may not understand precisely what the safe subsets of functionality are, or which subsets any real application will require. The underlying assumptions are that the setuid binary can be patched faster than the OS kernel, and that some experience may be required before a sufficient and minimal policy language can be defined. This paper observes, however, that almost all setuid binaries are using abstractions that are decades old with very well-understood policies, and that modern kernels have flexible infrastructures for security policy enforcement [48]. Thus, there is no compelling reason to prefer setuid binaries to dynamically configurable policies enforced by a kernel security module.

The contributions of this work are as follows:

- A study of the policies encoded in setuid-to-root binaries on current Linux systems, considering privilege from the perspective of the non-administrative user.
- Identifying a set of straightforward changes to Linux which would obviate the need to violate the least privilege principle on most systems. Our prototype reduces the trusted computing base of Ubuntu Linux by 12,717 lines.
- Evaluating these changes on Linux, demonstrating that setuid-to-root binaries can be deprivileged with minimal overheads, minimal changes to trusted code, and no loss of functionality for users. For instance, a Linux kernel compilation on Protego is less than 2% slower than on unmodified Linux.

Thus, this paper demonstrates that most deployed systems can uphold the Least Privilege Principle at minimal costs. We are continuing to investigate the long "tail" of infrequently installed setuid binaries; Section 5.4 surveys the expected effort to remove setuid-to-root altogether. We are also pursuing practical adoption of these techniques. For instance, the maintainers of the Debian `eject` package have agreed to changes that would deprivilege the `dmcrypt-get-device` binary.

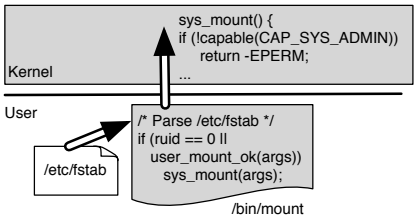
2. Protego Overview

Protego executes setuid-to-root binaries without privilege. Underlying a fairly wide range of packages (currently 82) containing setuid-to-root binaries, are a small number of system calls (8), system files, and devices that are either root-only, or have coarse subsets of functionality that are root-only. Protego centralizes the policies currently encoded in this disparate collection of trusted binaries by instead adding Linux Security Module (LSM) hooks (§3.2) that apply equivalent policies in the kernel. Protego is an extension of AppArmor on Linux 3.6.0.

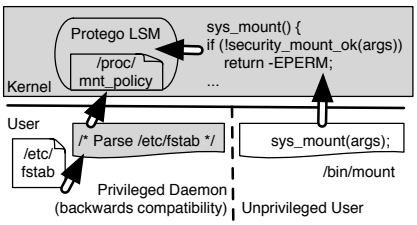
To explain the system design, we use the `mount` system call as a representative example. The `mount` system call grafts a new file system onto the file system directory tree at a given location. Similarly, the `umount` system call removes a file system from the directory tree. In general, changing the file system directory tree requires root privilege, as a malicious user might mount bad configuration files over `/etc` or even replace the system binaries in `/bin` or `/sbin`. Thus, the Linux kernel currently rejects any `mount` or `umount` call from processes without administrative privilege—namely the `CAP_SYS_ADMIN` capability, explained in §3.2.

Some mount-related binaries are `setuid` to root. The reason is to permit certain privileged operations without requiring administrator privilege, such as mounting a CD-ROM at `/dev/cdrom`. Operational constraints for these privileged functions are set by the administrator in `/etc/fstab` with the "user" or "users" option. The mount utilities, e.g., `mount` or `umount`, are then responsible for checking the real UID with which they were invoked: they are required to fail unless the file system and mount point match a "user" entry in `/etc/fstab`.

Protego is based on the observation that the kernel can just as easily perform these checks in an LSM. Both approaches are compared in Figure 1. For the `mount` system call, Protego keeps a whitelist of allowed user mountpoints in the kernel. If a process without `CAP_SYS_ADMIN` calls `mount`, the system call will only succeed if the arguments match the whitelist. Mount utilities no longer require administrative privilege when invoked by unprivileged users, as the policies are migrated to the kernel.



Linux `mount`.



Protego `mount`.

Figure 1. Comparison of the `mount` system call on Linux and Protego. Trusted components are in gray. Linux places trust in the `/bin/mount` binary to enforce policies specified in `/etc/fstab`; this binary is called by an untrusted user. The `mount` system call fails if the caller doesn't have the `CAP_SYS_ADMIN` capability (i.e., is not root). In Protego, a trusted daemon reads the policies from `/etc/fstab` and configures the Protego LSM through a file in `/proc`. Separately, an untrusted user can use `/bin/mount`, *or any other binary*, to issue a `mount` system call. The `mount` system call then calls an LSM hook to check this change against system policy.

This mount whitelist can be created by either the administrator by directly adding entries to a file in `/proc`, or, for convenience, we also provide a trusted daemon that reads and monitors `/etc/fstab`, propagating changes to the kernel via the `/proc` file. Protego provides two other files in `/proc` for configuration inputs using a simple grammar: a mapping of privileged ports to allowed application paths, and an `/etc/sudoers`-like syntax for delegation. Policies that do not take configuration parameters are simply hard-coded in the LSM. The underlying policy abstractions, concerns, and defaults derive from our study of current setuid-to-root binaries (Section 4).

Table 2 details the added or modified lines of code in Protego. The only code changes we made to most setuid utilities was removing checks that cause the binary to exit if the effective user id is not root—all policy checks are moved into the OS kernel and the utility runs without changing its effective user id to root. In a few cases, such as `vipw`, we modified these utilities to use different configuration files.

Backward compatibility. Protego modifies a few system configuration file formats. For example, Protego fragments the password database to better match policy and file system permissions; the monitoring database can keep the original database file and the new files synchronized for backwards compatibility. Although some previously-privileged applications must make changes to adopt the newer file formats, Protego maintains copies of the old formats for compatibility with other applications. Applications that did not previously require privilege are unmodified on Protego.

Component	Description	Lines
Kernel		
Linux	Additional LSM hooks, /proc filesystem interface.	415
Protego LSM module	Implement security policies, called by additional LSM hooks in Linux.	200
Netfilter	Extensions for raw sockets.	100
Trusted Services		
Monitoring daemon	Trusted process that monitors changes in policy-relevant configuration files. Required only for backwards compatibility.	400
Authentication utility	Trusted binary launched by the kernel to authenticate user sessions, password protected groups. Code refactored from `login` and `newgrp`.	1200
Utilities		
iptables	Extension for raw sockets.	175
vipw	Modified to edit per-user files instead of a shared database file.	+40
dmcrypt-get-device	Switch to /sys to read underlying device information.	43
mount/umount, sudo, pppd	Disable hard-coded root uid checks.	-25
	Grand Total Changed	2,598

Table 2. Lines of code written or changed in Protego, including kernel, trusted services, and command-line utilities.

The monitoring daemon is written with a Python library [4] based on Linux's `inotify` file monitoring framework [40]. The monitoring daemon is only required for backward compatibility.

Threat model. We assume that setuid binaries may have programming errors that are exploitable through inputs carefully crafted by an adversary. We assume the adversary is an unprivileged user of a system who aims to acquire enhanced privileges (one or more administrative capabilities, or root access, depending on the system).

Protego's goal is to minimize the privilege held by binaries executed by a non-administrative user. Even if one of these binaries is compromised, the user acquires no more privilege than she already had before executing the binary. Other privilege escalation attack vectors, such as vulnerable system daemons running as root or bugs in system calls, are beyond the scope of this paper.

3. Background

This section explains background on the setuid bit, related efforts to improve Linux security, and current setuid installation statistics.

3.1 The Setuid Bit

Setuid bit vs. system call. The namespace collision between the setuid permission bit and setuid system call can

lead to some confusion. The primary mechanism to raise privilege in Linux is the setuid permission *bit* (04000) in an inode's `stat` field. When a setuid binary is executed, the process executes as the binary's owner, regardless of which user `exec`-ed the binary. Some privileged daemons, such as `ssh`, may initially execute as root and then drop to an unprivileged user by using the setuid *system call*. Similarly, setuid-to-root binaries often bound the risk of privilege escalation by dropping root privilege after completing the last privileged system call, using the setuid system call. Unless otherwise specified, setuid in this paper refers to the bit.

Papers including "Setuid Demystified" explain how an application should use the `setuid` *system call* to drop its privileges [16, 44]. Our primary interest is complementary: eliminating privilege altogether in current setuid-to-root binaries, and thereby eliminating the need to securely drop privilege with the setuid system call.

Why is setuid-to-root needed? Administrators use setuid binaries to relax hard-coded policy decisions in the kernel, which are inconsistent with the desired system policy. Hecht et al. [25] observe that there are three categories of system calls: (1) unprivileged calls, (2) privileged calls, and (3) calls with privileged options. Modern Linux kernels generally enforce a stricter policy on calls with privileged options than administrators want, limiting application functionality. For example, the `mount` system call fails if the caller doesn't have administrative privilege—*even if the caller is only requesting options considered safe by system policy*. In the cases of `bind` and `open`, setuid is often used to allow an application to access a single port or file, but trusts the binary with access to *all* other ports or files. These point solutions implement the desired policy, but violate least privilege.

setgid. This paper focuses on the `setuid`-to-root binaries, although similar issues could arise with the `setgid` bit. We note that no Debian or Ubuntu packages currently install binaries that are `setgid` to root. The delegation framework we describe in §4.3 provides equivalent functionality to setuid and setgid-nonroot on Protego.

Eliminating setuid-to-root binaries. Several Linux distributions have hardening efforts that have reduced the number of setuid-to-root binaries [21, 45]. For instance, Ubuntu has eliminated roughly 30 setuid-to-root packages since 2008. These efforts have used the following major techniques:

- **Consolidation.** When several different packages perform similar tasks, developers create a shared setuid helper utility, such as the `sensible-mda` mail server utility.
- **File system permissions.** Utilities such as `at` write to logs and other protected system files, generally under the `/var` directory. Root privilege can be replaced with setuid non-root or setgid non-root by changing permissions on these files to an unprivileged user or group.
- **Capabilities.** Linux has a coarse capability model, which we explain next. Several utilities have replaced setuid with the similar `setcap` mechanism, which launches the binary with specific capabilities. Although setcap can replace setuid, several setuid-to-root binaries require capabilities tantamount to root.

These techniques are insufficient to enforce least privilege on all categories of current setuid-root binaries.

3.2 Capabilities, LSMs, and SELinux.

Capabilities in Linux are not pointers with fine-grained access control information, as commonly defined in capability-based operating systems [31, 39]. Linux divides root privilege into roughly 36 capabilities, called file system capabilities [18], which are roughly based on Trusted IRIX capabilities [26] and the POSIX.1e draft specification [36]. All uses of the term "capability" in this paper refer to Linux file system capabilities.

By default, Linux gives all capabilities to a process running as root. A hardened Linux variant can reduce the capabilities granted to the administrator for a given task, as well as limit the capabilities given to a setuid-to-root binary.

Capabilities are designed to enforce least privilege *on the administrator*, but are generally too coarse to enforce least privilege on unprivileged users. For instance, if the administrator is configuring a network interface, the configuration utility may only run with the `CAP_NET_ADMIN` capability. If the configuration utility is buggy and makes errant privileged system calls, the damage is limited to the network.

Continuing our example, `CAP_NET_ADMIN` is required by the setuid-to-root `pppd` binary so that unprivileged users may make very restricted changes to the system's routing tables, such as creating a route if it does not conflict with previously existing routes. Even if `pppd` executes only with the `CAP_NET_ADMIN` capability, if `pppd` is compromised, the unprivileged user has substantially escalated her privileges, gaining the ability to arbitrarily change routes, disable devices, set privileged socket options, enable multicasting, etc. A potential solution to this problem is to bestow finer-grained capabilities on trusted binaries.

Unfortunately, developers have failed to effectively manage 36 coarse capabilities in the Linux kernel. Most Linux developers are not security experts, but nonetheless are required to place capability checks throughout the kernel. When in doubt, developers use the `CAP_SYS_ADMIN` capability. As a result, over 38% of all capability checks in Linux require this capability. Even our initial `mount` example requires `CAP_SYS_ADMIN`. Moreover, this capability has become so permissive that it can acquire all other capabilities and is described as "the new root" [27]. Finally, the mapping of capabilities to privileged tasks is many-to-many. For instance, the X server requires 4 capabilities to set the video mode (`CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_SYS_RAWIO`, and `CAP_SYS_ADMIN`). Changing passwords requires 6 capabilities: `CAP_SYS_ADMIN`, `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_SETUID`, `CAP_DAC_READ_SEARCH`, and `CAP_FOWNER`. Linux capabilities do

Package	Ubuntu(%)	Debian(%)	Wt.Avg.(%)
mount	100.00	99.75	99.99
login	99.99	99.82	99.98
passwd	99.97	99.84	99.97
iputils-ping	99.87	99.60	99.85
openssh-client	99.54	99.48	99.53
eject	99.68	90.95	99.24
sudo	99.48	74.34	98.21
ppp	99.54	45.65	96.81
iputils-tracepath	99.78	13.06	95.39
mtr-tiny	99.54	11.79	95.10
iputils-arping	99.60	3.55	94.74
libc-bin	50.14	86.15	51.96
fping	27.70	12.42	26.92
nfs-common	9.76	82.89	13.46
ecryptfs-utils	11.64	0.72	11.08
virtualbox	10.56	7.78	10.41
kppp	10.11	4.97	9.85
cifs-utils	2.59	19.23	3.43
tcptraceroute	0.33	23.38	1.50
chromium-browser	0.48	8.49	0.89

Table 3. Percent of systems that install packages containing setuid-to-root binaries, as reported by the Debian and Ubuntu 'popularity contest' surveys. Average is weighted by the total number of systems reporting in each survey.

not enforce least privilege on the administrator, much less limit the risk of privilege escalation by unprivileged users.

The deeper problem with Linux capabilities is that they provide an insufficient language to express policies for unprivileged users. Linux capabilities require a subject-based policy ("allow if requester is root"), yet most `setuid`-root binaries export select, safe operations on a kernel abstraction to all users—an object-based policy. As a result, system security hinges on a cumbersome, error-prone translation of an object-based policy onto a strictly subject-based language.

Linux Security Modules (LSMs). LSMs encapsulate sophisticated security policies from the rest of the kernel development process by placing suitable access control hooks throughout the kernel [48]. Security experts can then implement more advanced policies, such as mandatory access control (MAC), by using these hooks to override the default, discretionary access control policies. These hooks are intended to be sufficient to implement any policy without making additional changes outside of the module. The exact number of hooks varies (184 in Linux 3.13.5); Protego adds additional LSM hooks for system calls that are currently hard-coded to check specific capabilities.

Security-hardened Linux variants. SELinux [32] and AppArmor [6] are implemented as LSMs, as is Protego. SELinux provides a powerful mandatory access control (MAC) and multi-level security (MLS) model for Linux, complete with role-based access control [22] and security type enforcement. AppArmor is also an MLS implementation, and the default on Ubuntu. AppArmor tends to enforce coarser policies than SELinux.

LSMs can carefully control when a process is given a capability, and can use an LSM hook to obviate a capability check for some, but not all system calls. For example, SELinux associates capabilities with roles, which can prevent a process from accumulating capabilities. SELinux can also replace the coarse capability checks in `bind` with an allocation of low-numbered ports to types.

SELinux requires considerable policy effort to expose safe functionality to non-administrator users. For instance, SELinux must carefully manage the `CAP_NET_RAW` capability and assign it to trusted binaries, such as `ping`. This does not enforce least privilege, as `CAP_NET_RAW` is coarser than `ping`'s safe functionality (§4.1). SELinux could enforce simpler and more precise policies by adopting Protego's strategy of considering safe functionality separately from fragmenting administrator privilege.

Tools such as VulSAN [15] analyze system attack surface, generating the path for an attacker to install a rootkit. In many cases, the path goes through a setuid or capability-enhanced program, even on SELinux or AppArmor.

3.3 Setuid Installation Statistics

To focus our efforts on removing privilege from the most commonly-installed setuid-to-root binaries, we studied installation statistics collected by the Debian and Ubuntu distributions. The first step was to identify all potentially installable setuid binaries in all Debian and Ubuntu 12.10 APT repositories using the Lintian reports [9]. Ubuntu adopts and repackages stable versions of Debian packages, and the differences between the distributions tend to be minor. 82 packages contain setuid-to-root binaries.

We obtained the rough frequency of installation from the popularity contest results for all the monitored Ubuntu [10] and Debian [8] systems, based on over 2.5 million systems (2,502,647 Ubuntu and 134,020 Debian).

Table 3 lists the 20 most frequently installed packages, with per-distribution percentages and an average weighted by number of installations of each distribution. We have completely investigated all popular packages through ecryptfs-utils—indicating that roughly 89.5% of sample systems could adopt Protego with no loss of functionality. The 62 packages not listed are installed by fewer than .89% of systems sampled; Section 5.4 summarizes the additional work we foresee in deprivileging the remaining packages.

4. Setuid Policy Study

This section presents a detailed study of the policies encoded in the 28 most commonly installed setuid-to-root binaries. This study identifies the system-level policy goal encoded in the binary, how the kernel policy for a specific system call or other interface is mismatched to the policy goal, how low-level mechanisms can be easily modified to efficiently and comprehensively enforce these policies in the kernel, and how Protego enforces these policies.

Interface	Used by	Kernel policy	System policy	Security concerns	Our approach
socket	ping, ping6, arping, mtr, traceroute6.-iputils	Creating raw or packet sockets requires CAP_NET_RAW.	Users may send and receive safe, non TCP/UDP packets, such as ICMP.	Raw sockets allow one to send both benign packets (e.g., ICMP) and packets that appear to come from socket owned by another process.	Allow any user to create a raw or packet socket, but outgoing packets are subject to firewall rules that filter unsafe packets.
ioctl	pppd	Only the administrator my configure modem hardware or modify routing tables.	A user may configure a modem (if not in use) and add routes that don't conflict with existing routes.	Protect the integrity of routes for unrelated applications.	Add LSM hooks that verify routes do not conflict with old rules when requested by non-root users.
	dmcrypt-get-device	Require CAP_SYS_ADMIN to read dmcrypt metadata.	Any user may read public portion of dmcrypt metadata (e.g., device set).	The same ioctl discloses both the physical devices and the encryption keys.	Abandon this ioctl for a /sys file that only discloses the physical devices.
bind	procmail, sensible-mda, exim4	Require CAP_NET_BIND_SERVICE to bind to ports < 1024.	Mail server should generally run without root privilege.	Prevent untrustworthy applications from running on "well-known" ports.	System policies allocating low-numbered ports to specific (binary, userid) pairs.
mount, umount	fusermount, mount, umount	Mounting or unmounting a file system requires CAP_SYS_ADMIN.	Any user may mount or unmount entries in /etc/fstab with the "user(s)" option.	Protect the integrity of trusted directories (e.g., /etc, /lib);	Add LSM hooks that permit anyone to mount a white-listed file system with safe locations and options.
setuid, setgid	polkit-agent-helper-1, sudo, pkexec, dbus-daemon-launch-helper, su, sudoedit, newgrp	Only allowed with CAP_SETUID.	Permit delegation of commands as configured by administrator, in some cases require recent reauthentication.	Require authentication and authorization to execute as another user.	Add LSM hooks that check delegation rules encoded in files like /etc/sudoers, and a kernel abstraction for recency.
Credential databases	chfn, chsh, gpasswd, lppasswd, passwd	Only root can modify these files (or read /etc/shadow).	A user may change her own entry to update password, shell, etc.	Prevent users from accessing or modifying each other's accounts.	Fragment the database to per-user or per-group configuration files, matching DAC granularity.
Host private ssh key	ssh-keysign	Only root may read the key (FS permissions)	Allow non-root users to sign their public key with the host key, disabled by default.	A user should be able to acquire a host key signature without copying the host key.	Restrict file access to specific binaries instead of, or in addition to, user IDs.
Video driver control state	X	Root must set the video card control state, required by older drivers.	Any user may start an X server.	An untrustworthy application could misconfigure another application's video state.	Linux now context switches video devices in the kernel, called KMS.
/dev/pts* terminal slaves	pt_chown	Root must allocate pts slaves on pre-2.1 kernels.	Users may create terminal sessions.	This utility has been obviated for 17 years, but is still shipped.	Ignore.

Table 4. System abstractions used by commonly installed setuid utilities on recent Debian and Ubuntu systems. The table identifies the common thread of inconsistent kernel policies and system policies for these abstractions, discusses the underlying security concerns, and how Protego unifies system and kernel policies.

In the interest of brevity, we do not discuss each binary in detail, but rather organize our discussion around each interface that requires administrative privilege. Table 4 summarizes these results. The Protego design is guided by two major themes from this study:

- **Express and enforce object-based policies.** The majority of these binaries encapsulate sensible policies that do not map onto subject-based checks (e.g., "is this user x?"), but that can be easily expressed as object-based policies (e.g., "may any user take this action on this object?"). For instance, several utilities enforce fine-grained access control on network ports, and enforce policies for raw sockets based on the protocol type (§4.1).
- **Interface designs can thwart least privilege.** In several cases, poor interface design can require applications to have more privilege than necessary. For instance, the `dmcrypt-get-device` utility uses a privileged `ioctl` to report the physical device underneath an encrypted block device; additional privilege is required because this ioctl also discloses the private key. A more subtle example of this point is the division of labor between user and kernel in the X window server (§4.5). Where needed, Protego adjusts these interfaces.

Several of the issues raised in this study could be addressed in more than one way, and some already have point solutions in another OS. This section strives to delineate the essential requirements of any solution from the particular solution implemented in the Protego prototype, as well as cite existing alternative point solutions. To the best of our knowledge, however, no system has comprehensively addressed all of the issues requiring setuid-to-root binaries.

4.1 Network

Three networking tasks require privilege: creating a raw or packet socket, the point-to-point protocol (PPP), and binding a socket to a TCP or UDP port less than 1024.

4.1.1 Raw and packet sockets

When applications program with TCP or UDP sockets, the kernel encapsulates the application-level payload with the appropriate headers. In the rare case that an application needs to send messages with a protocol the kernel doesn't implement, the application must create most of the packet headers itself, using a *raw* or *packet socket* [5]. The difference between raw and packet sockets is that raw sockets provide the IP layer and MAC layer headers, whereas a packet socket only implements the MAC layer headers.

Linux requires the `CAP_NET_RAW` capability to create a raw or packet socket, because an application can also create packets that appear to come from another application. However, a number of setuid-to-root binaries, such as `ping`, allow unprivileged users to send safe packets over raw sockets.

A more precise, object-based policy, would specify which types of outgoing packets are acceptable from unprivileged users on raw sockets. We observe that the set of all safe packets exported by setuid-to-root binaries can be easily encoded as a whitelist in any packet filtering framework, such as Linux's netfilter/iptables [2], or BSD's Berkeley Packet Filters (BPF) [33]. For instance, some BSD variants use BPFs to sandbox binaries in a similar manner, such as ensuring the DHCP server sends DHCP packets only to limited addresses [43].

The Protego prototype allows unprivileged programs to create raw sockets, with the caveat that these unprivileged raw sockets are subject to additional netfilter rules. The default rules are based on the studied setuid-to-root binaries, but the rules may be changed by the administrator through the `iptables` utility. Protego implements this with modest extensions to the Linux netfilter framework, as well as adding an LSM hook to the `socket` system call.

In Protego, a compromised network utility cannot spoof packets from a TCP or UDP socket, unlike all current Linux variants. Also in contrast to Linux, Protego allows any unprivileged user to create her own enhanced `ping` utility, as long as it conforms to system security policy.

4.1.2 Point-to-point protocol (PPP)

PPP is a protocol used to establish connections over modems, including dial-up and cellular networks. In most Linux systems, the service that manages a PPP connection (`pppd`) requires root privilege for two tasks: configuring modem hardware and, potentially setting system routing tables to relay IP traffic through a PPP link. The `pppd` binary is setuid root so that it can be launched on demand, because, unlike ethernet, most PPP connections are not constantly active.

When `pppd` is launched as a user other than root, only certain safe configuration options are accepted, such as compression and congestion control session parameters. An administrator can also configure `pppd` to allow unprivileged users to add system routes over a ppp connection, but only if the new address ranges were not previously reachable. If a PPP connection duplicates an existing route, a user may only create a `tty` that communicates with the remote point.

We add LSM hooks to the appropriate `ioctl` system calls for configuring routing tables and modem options. Specific policies are mined from the ppp configuration file `/etc/ppp/options`. We also changed the default file system permissions on `/dev/ppp` to be more permissive, replacing a capability check with device file permissions.

We verified that `pppd` works without root privilege by connecting two machines over a crossover serial cable, such that one serves as an internet gateway to the other. Both machines ran `pppd` without root privilege, both were able to create routing table entries, and the non-gateway machine was able to connect to remote websites.

4.1.3 Bind

Creating a socket that listens to a TCP or UDP port less than 1024 requires root or the `CAP_NET_BIND_SERVICE` capa-

bility. For this reason, many network services run with root privilege, at least temporarily, to set up a listening socket.

This policy reflects the notion that an administrator should be involved in setting up a service that listens on a well-known port. For instance, one expects that a web server listening on port 80 is endorsed by the administrator, but not if it is listening on port 8080. However, any privileged application can open any privileged port; for instance, a malicious web server can also act as an email or DNS server.

The system policy goal is that specific ports should be allocated to specific application instances. Protego uses a tuple of (binary path name, user ID) to represent an application instance, and a simple policy configuration file, `/etc/bind`, which maps each TCP or UDP port less than 1024 to an application instance. Each port may map to only one application instance.

Our strategy is a simplified version of SELinux's approach, which allocates ports to types. Using types to specify an application instance is sufficient, but not necessary; eliminating privilege escalation should not hinge on adopting SELinux. Similarly, Berkeley Packet Filters can also be used to delegate access to a privileged port; a centralized configuration has the advantage of easier auditing.

4.2 Mount

§ 2 explains our approach to the mount utilities as a motivating example; we will not repeat this for space. We validated that unprivileged users can mount `user` entries in `/etc/fstab` but not other file systems.

An alternative point solution used by many Linux and Unix systems is a trusted daemon (e.g., `automounter` [3]) that monitors accesses to whitelisted mount points, and automatically mounts them in response to attempts to access the device. Any design that allows unprivileged users to access whitelisted mount points is sufficient; the Protego design adds only 258 lines of trusted code to the system, whereas the Linux automounter implementation increases the TCB by 21,674 lines, including a 79 line kernel patch.

4.3 UID Switching and Delegation

A process changes its user and group IDs using a family of system calls including `setuid` and `setgid`, which require the `CAP_SETUID` or `CAP_SETGID` capabilities. These capabilities give a process the ability to assume any user's or group's id. Utilities such as `sudo` are setuid root so that they start with administrator privilege, validate that the invoking user is allowed to switch to the new user, and then call `setuid` to drop privileges.

Because our interest is in enforcing least privilege on non-administrator users, we focus on *lateral* moves, where one unprivileged user (Alice) acts for another (Bob). Most delegation utilities, such as `sudo`, violate the principle of least authority by giving Alice the privilege to run as *any* user via a setuid root binary, and then dropping back down to Bob with the setuid system call.

Ideally, Alice's `sudo` process should never be able to switch to any uid other than Bob's, and only in ways Bob has authorized her to act for him. If sudo enforced least privilege on Alice, even if Alice compromises her sudo process, she should never be able to switch to an unrelated user, say Charlie. Obviously, in many cases, `sudo` is used to transition to root; even in this case, root privilege should only be granted to the process *after* all checks have succeeded, not before. These policies are only possible if the kernel, not a user application, validates `setuid` system call transitions.

Taking `sudo` as a representative example, there are two checks that require a trusted agent:

1. **Authentication:** Either the current or target user's password should be entered and checked. Utilities like `sudo` check the calling user's password to ensure that another person hasn't sat down at an unattended terminal. `sudo` only checks the password if a password has not been entered on the terminal in the last 5 minutes. In contrast, utilities like `su` ask for the target user's password as both authentication and authorization to act as the target user.
2. **Authorization:** The administrator may configure `sudo` to only allow a user to issue specific commands as another user. For instance, Alice may allow Bob to issue the `lpr` command to print with her credentials, but `sudo` will not allow Bob to directly execute any other binaries as Alice. Specifying a particular command also requires limiting inheritance of environment variables or open file descriptors, to ensure integrity of the delegated command.

An untrusted binary should not be charged with either.

Rather than check for administrator privilege on a `setuid` system call, Protego enforces the policies collected from the `/etc/sudoers` configuration file and the configuration files in the `/etc/sudoers.d/` directory. Policies currently encoded in setuid binaries are explicated in additional `/etc/sudoers` rules. Similar to `/etc/fstab`, the monitoring daemon parses these files, watches for changes, and sets the kernel policies accordingly. Protego also requires that any user issuing a `setuid` system call be recently authenticated, unless specifically countermanded by a `sudoers` policy with the `NOPASSWD` directive. We shift the validation of command-line arguments to the kernel.

The Protego kernel tracks the last authentication time in the `task_struct` of each process. If a `setuid` system call is issued without a recent authentication of the current user, a trusted authentication service temporarily takes over the terminal and asks for the user's password. This authentication service is refactored from the `login` source code. This service can also request the password of another user or group, according to system policy.

A challenge in restricting `sudo` privilege to a given binary is that policy enforcement must span two system calls: `setuid` and `exec`. To achieve least privilege, the process privileges must not change before the `exec` system call. To address this, we slightly change the behavior of `setuid` for

restricted UID transitions. If a process could make an `exec` call with at least one permissible binary, the `setuid` system call's return code (0) will indicate success to the application, but the system call will set a field in the task's security metadata indicating a pending setuid-on-exec and storing the pending user. When a process with the setuid-on-exec field set issues an `exec` call, Protego hooks the `exec` system call and checks whether the requested binary is allowed as the pending user. The authentication service may also ask for the target user's password at this point. If the user is not authorized to `exec` the requested binary as the target user, the `exec` will fail with a permission denied error. This change in error behavior is difficult to avoid when enforcement must span two system calls; in practice, our target utilities have worked correctly despite this change. We also add appropriate restrictions on inheritance through setuid transitions, except where explicitly permitted in the sudoers policy.

In our example of Alice acting for Bob, a setuid-to-Bob binary can provide the desired user transition. In practice, tools like `sudo` are preferred because `sudo` is more flexible, centralizes system policy in an easy-to-audit location, and sanitizes inputs and environment variables to reduce the application's attack surface. An alternative approach to implementing a least-privilege `sudo` might generate a set of setuid-nonroot binaries that encode the system policies, similar to capability amplification in Hydra [49].

Protego encodes the policies of a wide range of delegation utilities as extended sudoers rules, including su, sudoedit, dbus, policykit, and newgrp. For instance, `newgrp` exports password-protected groups; this functionality can be encoded by requiring the user authenticate on a setgid system call that requests certain groups. We omit full details of these utilities for brevity.

4.4 File System Permissions

Several setuid-to-root binaries require root in order to work around inconvenient file system permissions. For instance, mail servers use root privilege to access configuration files, such as `.forward`, even if the user's permissions otherwise block access to the file. Similarly, most of the software for managing one's local account, such as `passwd` and `chsh`, requires access to a single record in a database file, but the kernel only enforces access control at the file granularity.

In order for a user to change her password or default shell, she should be able to modify her own entry of the database files. However, if Alice can modify Bob's database entry, Alice can access or modify Bob's account—a clear security problem. Thus, the shared database files can only be written by root. In terms of capabilities, a process must acquire `CAP_SYS_ADMIN`, `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_SETUID`, `CAP_DAC_READ_SEARCH`, and `CAP_FOWNER` to complete this operation—undermining the goal of substantially limiting administrative privilege with capabilities. To permit users to modify their credentials and preferences, `passwd`, `chsh`, and similar utilities are setuid and validate that the update does not corrupt the entire database.

To modify one's account with least privilege, the system must enforce access control at the granularity of a user's record, not the entire database. Protego splits the shared database files into per-account files. For instance, we split `/etc/passwd` into one file per-user under `/etc/passwds/`. Each file has permissions `rw-------`, owned by the user it defines and the parent directory `/etc/passwds/` has permissions `rwxr-xr-x` owned by user root and group root so that unprivileged users cannot add new users to the system. For backward compatibility, a trusted daemon monitors the per-user files and updates the legacy `/etc/passwd` file (§2). A similar approach is taken with `/etc/group` and `/etc/shadow`.

Network directory servers, such as OSX's Directory Server [11] or OpenLDAP [1], already enforce record-level access control. However, such a mechanism is generally not available for the local system accounts, and adds substantially more code to the system TCB (Protego changes 240 LoC, whereas OpenLDAP Version 2.8 is 175,368 LoC).

In the case of mail delivery problems, we advocate sufficient warnings in the log to diagnose delivery problems resulting from incorrect file permissions.

We note that users may desire finer-grained access control to prevent their password hash from leaking from their `/etc/shadows` file. Protego mitigates this risk by requiring the user to reauthenticate before reading the shadow file using the mechanisms described above (§4.3). The shadow file handle may not be inherited (`CLOSE_ON_EXEC`).

4.5 Interface Design

A small number of utilities require root privilege because the kernel's interface design forces otherwise unnecessary trust on the utility. In many cases, non-security reasons naturally lead to more sensible interfaces and obviate the need for trusted binaries. For instance, very old versions (before 2.1, or 1996) of the Linux pseudo-terminal implementation required applications to allocate slave terminal devices, and thus required a trusted binary to prevent applications from interfering with each other. Ultimately, allocating slaves in the kernel was simpler all around, and obviated the need for a trusted `pt_chown` utility.

Similarly, the X window server is setuid to root because the X server may need to both *manage* the graphics hardware as well as *display* the user's composite desktop. In practice, simply drawing a screen image to a frame buffer does not require any administrator privileges. Root privilege is required to configure and context switch the video hardware, e.g., setting the refresh rate, screen resolution, and, most importantly, configuring which application's frame buffer should be displayed on the screen [7].

In practice, video card management has been migrating from the X server into the kernel. When the X server manages the video card, developers find writing context switch-

ing code for a range of video hardware cumbersome. The video configuration changes when a user switches from one X server to another, or from the X server to a text console (via Ctrl-Alt-F1). The X server must correctly save and restore all state for all supported video cards. Thus, X and video driver developers have decided that a more sensible division of labor is for the OS kernel to manage context switching the video card. Linux now has a standard interface for device drivers to save and restore device state, called Kernel Mode Setting (KMS) [7], introduced in Linux 2.6.29.

KMS eliminates the need for X to run as root [19], and is being adopted by all major device manufacturers. Device drivers with KMS support have been written for chipsets manufactured by Intel, ATI (radeon), Nvidia (nouveau driver), and others. Nvidia's closed-source drivers have recently started adopting KMS [30]. We have verified this on our own machines using the Nouveau driver and running an X server binary that is not setuid to root.

An interesting contrast is that, unlike many setuid binaries, X and the Linux video drivers are still under very active development. The same decision that led to a needless attack surface has caused enough practical problems that developers are rectifying the problem for non-security reasons.

4.6 Limitations and Discussion

Our study covers a representative sample of setuid-to-root binaries, and we expect that Protego can easily support most of the remaining setuid-to-root binaries. We note that the Protego prototype allows an administrator to reenable the setuid bit if necessary. If the setuid bit is actually removed from all other binaries when the bit is reenabled on the system, the only marginal difference will be that the one unsupported binary is added to the system's trusted computing base.

The most likely situations where the setuid bit may be required are new kernel interfaces where the desired policy is not well understood. For instance, Linux has been gradually adding support for sandboxing with restricted namespaces [28], beginning with version 2.6.23. Until version 3.8, the security implications were not sufficiently understood, and sandboxing utilities, such as chromium-sandbox, had to run setuid-to-root. The security implications are now better understood, and newer kernels allow unprivileged users to use safe namespace configurations.

The main weakness of the Protego design is that its tools for mitigating information leaks are limited compared to SELinux, but more powerful than unmodified Linux. Two setuid binaries need to read sensitive data: `ssh-keysign` and password changing utilities. In these cases, a measure of trust is unavoidable. Protego develops fine-grained delegation rules (§4.3) that allow only these trusted binaries to read specific sensitive files. These restrictions are not as strong as SELinux roles [22, 32]. For example, a keysigning role might take away network access and write permission to any file handle other than a pipe to the parent. In comparison, Linux allows *any* program with root privilege to read these files. Protego's contributions are orthogonal to SELinux's, and could be adopted by SELinux to mutual benefit.

Test	Linux	+/-	Protego	+/-	% OH
lmbench					
syscall (us)	0.04	0.00	0.04	0.00	0.00
read	0.09	0.00	0.09	0.00	0.00
write	0.09	0.00	0.09	0.00	0.00
stat	0.34	0.00	0.33	0.00	-2.94
open/close	1.17	0.09	1.17	0.09	0.00
mount/umnt	525.15	18.82	531.13	19.44	1.13
setuid	0.82	0.09	0.83	0.06	1.22
setgid	0.82	0.09	0.83	0.09	1.22
ioctl	2.76	0.45	2.78	0.48	0.72
bind	1.77	0.10	1.81	0.11	2.25
sig install	0.10	0.00	0.10	0.00	0.00
sig overhead	0.70	0.00	0.70	0.00	0.00
prot. Fault	0.19	0.00	0.19	0.00	0.00
fork+exit	159.00	0.99	158.00	0.40	-0.63
fork+execve	554.00	1.86	573.00	1.81	3.43
fork+/bin/sh	1360.00	1.33	1413.00	1.72	3.90
0KB create	5.57	0.39	5.43	0.37	-2.51
0KB delete	3.93	0.59	3.79	0.73	-3.56
10KB create	11.00	0.13	10.80	0.10	-1.82
10KB delete	5.90	0.36	5.85	0.57	-0.85
AF_UNIX	9.30	0.00	9.69	0.00	4.19
Pipe	6.73	0.00	6.88	0.00	2.23
TCP connect	18.00	0.00	18.55	0.00	3.05
Local TCP lat	19.63	0.00	20.87	0.00	6.32
Local UDP lat	16.70	0.00	17.90	0.00	7.19
Rem. UDP lat	543.60	0.00	578.30	0.00	6.38
Rem. TCP lat	588.10	0.00	631.50	0.00	7.38
BW (MB/s)	5316.60	0.00	5170.69	0.00	2.74
Postal Benchmark for Exim server					
Messages/min	258.64	3.33	258.75	3.09	0.04
Kernel Compile					
time(s)	764.41	4.36	775.39	5.28	1.44
Apache Bench					
Time per request (ms, lower is better)					
25 conc. reqs	0.28	0.01	0.29	0.01	3.57
50 conc. reqs	0.26	0.00	0.27	0.00	3.85
100 conc. reqs	0.25	0.01	0.26	0.01	4.00
200 conc. reqs	1.13	0.02	1.16	0.02	2.65
Transfer rate (Kbps, higher is better)					
25 conc. reqs	6781.04	134.14	6506.29	157.34	4.05
50 conc. reqs	7375.21	253.56	7083.63	248.34	3.95
100 conc. reqs	7342.15	206.45	7051.54	236.78	3.96
200 conc. reqs	1642.90	106.46	1599.55	113.92	2.64

Table 5. Protego overheads compared to Linux with AppArmor. Unless otherwise noted, tests measure execution time in microseconds (lower is better). A few tests measure bandwidth in MBps or Kbps, where higher is better.

5. Evaluation

This section aims to answer the following questions:

1. What is the cost (in execution time and network throughput) of moving setuid policies into the kernel? In particular, what cost is imposed on applications that do not use any privileged functionality?
2. How does Protego affect overall system security?

3. Are Protego functionality and policies identical to unmodified Linux?
4. What effort would be required to deprivilege the remaining 67 packages containing setuid-to-root binaries?

Our baseline is an unmodified Linux 3.6.0 kernel configured to use AppArmor and iptables with no firewall rules on Ubuntu 12.04. Protego was refactored from Linux 3.6.0. All measurements were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU. The test machine has 4 GB of memory and a 250GB, 7200 RPM SATA disk.

5.1 Performance Overheads

We measure the performance cost of Protego policy enforcement on applications that do not require any privilege on Linux with both application benchmarks and the lmbench microbenchmark suite, version 3.0-a9. We note that within a run, we use the standard lmbench configuration, which includes a number of iterations scaled appropriately to the test case. We report the mean and 95% confidence intervals. We also measure Linux kernel 3.6.6 compilation time and network performance using the ApacheBench benchmark, version 2.3, exercising an Apache web server, version 2.2.16.

We note that most applications which use the `setuid` bit are interactive, and thus difficult to meaningfully benchmark. The most interesting exception to this are the mail servers, so we include measurements of exim4 server using the Postal [17] benchmark. We also extend lmbench with 5 additional tests that exercise the system calls we modified.

In general, the overheads of Protego are low, ranging from 0–7.4%. Table 5 lists the measurements of Protego, as well as overhead relative to Linux. For many system calls, Protego has no effect on the behavior and performance is comparable. In the case of `exec`, for instance, Protego adds 5.78% overhead to enforce setuid policies. A few microbenchmarks show small performance improvements commensurate with the confidence intervals, which we believe are noise. At the macro-benchmark level, such as a Linux kernel compile, Protego overhead is only 1.44%.

In order to enforce policies on raw network packets, we add additional netfilter rules on all outgoing packets. To measure the impact on unprivileged applications, we compare to netfilter with no rules configured, and the overhead ranges from 2–4% for the standard ApacheBench web service benchmark. These results indicate that the performance overheads are acceptable.

5.2 Security Evaluation

It is difficult to quantify the change in risk of any change to a system interface and enforcement mechanism. To evaluate Protego we consider two rough indicators: the net change in the trusted computing base and whether historical vulnerabilities in `setuid` binaries would occur in code that is now de-privileged, or in code that moved into the OS kernel.

Utilities	Total CVEs	Priv. Esc.	CVE Identifiers
ping	84	4	1999-1208, 2000-1213, 2000-1214, 2001-0499
traceroute	26	2	2005-2071, 2011-0765
mount,umount	114	2	2006-2183, 2007-5191
mtr	4	3	2000-0172, 2002-0497, 2004-1224
sendmail	84	2	1999-0130, 1999-0203
exim	21	2	2010-2023, 2010-2024
sudo	61	5	2001-0279, 2002-0043, 2002-0184, 2009-0034, 2010-2956
sudoedit	3	1	2004-1689
newgrp	7	6	1999-0050, 2000-0730, 2000-0755, 2001-0379, 2004-1328, 2005-0816
passwd	87	1	2006-3378
passwd, su	-	1	2003-0784
su	31	2	2000-0996, 2002-0816
chsh, chfn, su, passwd	-	1	2002-1616
chsh, chfn	10	2	2005-1335, 2011-0721
dbus	22	1	2012-3524
pkexec, policykit	24	2	2011-1485, 2011-4945
X	33	2	2002-0517, 2006-4447
capabilities	7	1	2000-0506

Table 6. Historical vulnerabilities in setuid-to-root binaries (total, and those that lead to privilege escalation). In Protego, these utilities and the vulnerable code would be deprivileged and would not lead to a privilege escalation. Dashes are placed in the "Total" column for a CVE that spans multiple packages; the package totals are reported in other rows.

Trusted Computing Base. We first carefully measure the change in privileged lines of code in the previously `setuid` binaries. At a high level, Protego adds 715 lines of policy checking code to the Linux kernel, 400 lines of code in Python which monitors certain configuration files for changes, and a 1,200 line authentication utility. The authentication utility is refactored from existing trusted code in `login` and `newgrp`. To balance this number, we also measure the lines of trusted binary code that no longer execute with privilege—a total of 15,047. We are careful to report a conservative estimate—ignoring whitespace, comments, standard libraries, and any code that would have executed after dropping privilege in the original code. Thus, Protego decreases the lines of trusted code by at least 12,732.

One potential concern is that Protego (conceptually) migrates policy enforcement code from applications to the kernel. We hasten to note that the policy enforcement code in the kernel is only 200 lines of straightforward C code. Protego also expands the purview of policies enforced in the kernel; these concerns are localized to our LSM, and generally make existing kernel policies more precise.

Similarly, one might be concerned about adding the user-level authentication service and monitoring daemon to the trusted computing base. We note that authentication code is trusted in both systems, the only difference is how it is invoked. Parsing configuration files can introduce new vulner-

Binary	Coverage %	Binary	Coverage %
chfn	94.4	sudo	90.1
chsh	92.7	sudoedit	90.9
gpasswd	91.3	mount	94.1
newgrp	93.5	umount	92.5
passwd	91.0	ping	96.2
su	92.2		

Table 7. Gcov coverage of command-line setuid binaries

abilities, but the risk is small, as the files are simple. This daemon is written in a managed language with regular expressions (Python) to minimize the risks commonly associated with input parsing. The monitoring service could be eliminated by changing additional legacy code. The code we add to the TCB is short, simple, and easily audited.

A more subtle issue in selecting a configuration format is the risk of misconfiguration. Whenever possible, we chose to use legacy configuration files administrators would find familiar. This choice is debatable and largely incidental to the Protego design; one could just as easily have a single policy file, as SELinux does, and perhaps manage this with some application to assist the administrator, all using the same underlying `/proc` interfaces to the Protego LSM.

Historical Vulnerabilities. We surveyed the Common Vulnerabilities Database [35] entries for the 28 binaries this paper studies. We found 618 total vulnerabilities over the lifespan of these utilities—40 of which led to acquisition of root privilege, summarized in Table 6.

We manually analyzed these 40 privilege escalation vulnerabilities and determined that these executed in code that now runs without privilege in Protego. Most of these vulnerabilities exploit buffer overflows, format strings, or environment variables. Moreover, these vulnerabilities are not in code substantially similar to the parsing, policy checking, or authentication code that we have added to Protego's TCB.

This data indicates that the probability of new privilege escalation vulnerabilities is relatively low, but the overall risk is still substantial. Most of the credit for the low probability of privilege escalation belongs to efforts to drop privilege after the privileged system calls have executed. Nonetheless, our analysis indicates that kernel support in Protego can further reduce the risk of privilege escalation.

As code matures the probability of exploitable bugs decreases. Although the most popular packages are generally quite old, Ubuntu has added 21 setuid-to-root binaries based on new code over the last 3 years, despite a net reduction in setuid binaries. The long-term goal of this project is to completely obviate the need for new setuid-to-root binaries, and their considerably higher risk of exploitable bugs.

5.3 Functional Testing

In addition to manual functionality tests, we validate that Protego behaves equivalently to unmodified Linux with exhaustive test scripts for setuid command line utilities. We validate that the utilities have the same output and effects on both systems. We also use `gcov` [24] to measure the test coverage for the command-line utilities in Table 7, which is always above 90%. Although exercising nearly all code paths does not necessarily mean all inputs are handled equivalently on both systems, one can infer that the functionality and policy enforcement is very likely to be equivalent.

Interface	No. of Setuid Binaries
socket	14
bind	23
mount	3
setuid,setgid	24
Video driver control state	13
chroot/namespace	6
miscellaneous	8

Table 8. System abstractions used by setuid binaries in packages not included in the Section 4 study. Abstractions below the double line need to be addressed in future work.

5.4 Toward Zero Setuid-To-Root Binaries

This subsection surveys the remaining 67 packages (91 binaries) in Ubuntu Linux to assess how many can likely be deprivileged on Protego and how many will require different approaches. We note that this survey is preliminary, based on the documentation, and we have not tested these on Protego.

Table 8 groups the remaining binaries by the underlying interfaces that require privilege. We observe that 77 use interfaces already addressed by Protego, but may require refinement to the policies currently enforced. The remaining 14 binaries require privilege for:

- **Namespaces** (6). §4.6 explains that namespaces no longer require privilege in Linux kernel versions 3.8 and higher.
- **System administration** (3). Three binaries use privilege to reboot the system, load kernel modules, or configure the network. Some may be able to use PolicyKit or sudo (§4.3), but others may require additional consideration.
- **Open a custom device** (5). Virtualbox includes a kernel-level virtual machine monitor, which exports a custom device to 5 setuid binaries. These applications and the kernel module are tightly coupled, and additional work will be required to identify a sensible policy for this device.

Thus, we are optimistic that a few additional policy abstractions can complete our ongoing effort to obviate the need for all setuid-to-root binaries.

6. Related Work

This section surveys related efforts to reduce the privilege of setuid binaries, which generally speaking, either enforce least privilege on the *administrator*, but not regular user, or simply remove functionality from users.

Much related work on setuid binaries attempts to mitigate the risk of privilege escalation attacks by allocating privilege only to portions of a program (also known as privilege bracketing or privilege separation). Systrace [37] mitigates privilege escalation attacks by localizing privilege in setuid bina-

ries to a single system call (e.g., the `socket` call in `ping`). Executable based access control [14] specifies which files a trusted binary may open; this design cannot prevent an errant `passwd` utility from changing other users' passwords, nor does it restrict privileged system calls unrelated to files. Protection domains within a setuid binary have also been proposed to preventing exploits in unprivileged operations from leaking into privileged code [41].

Secure Xenix [23, 25], and other secure Unix variants [13, 29, 46, 47] developed modern best practices for enforcing least privilege on the administrator: fragmenting administrative privilege into roles or capabilities, restricting the ability to create more setuid binaries, and removing the setuid bit if a binary is overwritten. Limiting the risk of exploiting a setuid binary is complimentary to Protego's goal of eliminating the need for setuid binaries.

Plan 9 eliminates setuid binaries by making every OS resource a file or file system, and by prolific use of fine-grained capabilities [20]. For instance, Plan 9 represents all networking abstractions as files with capabilities, whereas Protego enforces finer-grained network security policies.

Although Windows has a richer access control model than Linux users and groups [42], Windows adopts similar practices for some resources, such as requiring Administrator privilege to create a raw socket [34].

Finally, Bastille [12] simply removes the setuid flag and supported functionality from many utilities. For instance, non-privileged users cannot mount a USB drive, ping another computer, or use the `traceroute` command—functionality that could be safely reinstated on Protego.

Namespaces. Linux 3.8 allows unprivileged sandboxing applications, such as `chromium`, to create sandboxed network, mount, user, and process namespaces [28].

Namespaces are designed for isolating untrusted binaries, and are simply the wrong tool for enforcing least privilege when accessing shared system abstractions. Inside of a sandbox, a process can appear to have any capability, but any externally visible operations are subject to the original user's privilege. For instance, network namespaces allow an unprivileged process to access a fake network interface, and send ICMP packets within a fake network with no routes to the outside world. The major caveat is that any connection to the outside world requires an agent outside of the sandbox with the appropriate capabilities. In our network example, the agent would still need the `CAP_NET_RAW` capability to send ICMP packets out of the sandbox. In contrast, namespaces cannot safely allow access to shared system resources, such as `passwd` updating the password database.

7. Conclusion

This paper presents a study of setuid-to-root binaries on modern systems, yielding insights into the nature of least privilege, especially that one should consider the administrator and user separately. There is an interplay between the division of labor between the kernel and userspace that directly impacts the need for trust; trusted binaries are often compensating for a design flaw in the system interface. Protego demonstrates techniques that can eliminate this long-standing attack surface without sacrificing functionality.

Acknowledgements

We thank the anonymous reviewers, our shepherd, Ted Wobber, Rob Johnson, and Vyas Sekar for insightful comments on earlier drafts of this paper. Ujwala Tulshigiri contributed to the Protego prototype implementation. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, and the Office of the Vice President for Research at Stony Brook University.

References

[1] The openldap project. `http://www.openldap.org/project/`, Aug. 1998.

[2] netfilter. `http://www.netfilter.org/`, Dec. 2001.

[3] The automounter autofs. `http://www.autofs.org/`, Sep. 2002.

[4] Py-notify. `http://home.gna.org/py-notify/`, Dec. 2008.

[5] SOCK_RAW Demystified. `http://www.sock-raw.org/papers/sock_raw`, May 2008.

[6] AppArmor. `http://wiki.apparmor.net/index.php/Main_Page`, Jun. 2011.

[7] Kernel mode setting. `https://wiki.archlinux.org/index.php/Kernel\_Mode\_Setting`, Jan. 2011.

[8] Debian Popularity Contest. `http://popcon.debian.org/by_inst`, Feb. 2013.

[9] Lintian Reports : setuid-binary. `http://lintian.debian.org/tags/setuid-binary.html`, Feb. 2013.

[10] Ubuntu Popularity Contest. `http://popcon.ubuntu.com/by_inst`, Feb. 2013.

[11] Apple. Mac OS X Server V10.6 - Open Directory Administration. White Paper, Apple: `http://manuals.info.apple.com/MANUALS/1000/MA1180/en_US/OpenDirAdmin_v10.6.pdf`, August 2009.

[12] Securing Debian Manual: Bastille Linux. `http://www.debian.org/doc/manuals/securing-debian-howto/ch-automatic-harden.en.html\#s6.2`, Apr. 2012.

[13] M. Bishop. Managing Superuser Privileges under UNIX. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1986.

[14] B. Bringert. Executable based access control. Technical report, Chalmers University of Technology, 2003.

[15] H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems.

In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[16] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *Proceedings of the USENIX Security Symposium*, pages 171–190, 2002.

[17] R. Coker. Benchmarking Mail Relays and Forwarders. In *OSDC Conference*, 2006.

[18] J. Corbet. File-based capabilities. `http://lwn.net/Articles/211883/`, November 2006.

[19] J. Corbet. Rootless X. `http://lwn.net/Articles/341033/`, July 2009.

[20] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in Plan 9. In *Proceedings of the USENIX Security Symposium*, pages 3–16, 2002.

[21] Fedoraproject Wiki: Features/RemoveSETUID. `https://fedoraproject.org/wiki/Features/RemoveSETUID`, Apr. 2011.

[22] D. Ferraiolo and D. Kuhn. Role-Based Access Control. In *15th National Computer Security Conference*, pages 554–563, 1992.

[23] V. Gligor, C. Chandersekaran, R. Chapman, L. Dotterer, M. Hetch, W.-D. Jiang, A. Johri, G. Luckenbaugh, and N. Vasudevan. Design and implementation of Secure Xenix. *IEEE Transactions on Software Engineering*, SE-13(2):208 – 221, Feb. 1987.

[24] N. M. Guire. Linux kernel gcov - tool analysis. `http://dslab.lzu.edu.cn:8080/docs/2006summerschool/team1/teama/Documentation/howtos/der_herr_gcov.pdf`, 2006.

[25] M. Hecht, M. Carson, C. Chandersekaran, R. Chapman, L. Dotterrer, V. Gilgor, W.-D. Jiang, A. Johri, G. Luckenbaugh, and N. Vasudevan. UNIX without the Superuser. In *Proceedings of the USENIX Security Symposium*, pages 243 –256, June 1987.

[26] K. Johnson, J. B. Zurschmeide, J. Raithel, T. Schultz, and B. Tuthill. IRIX admin: Backup, security, and accounting. Technical report, Silicon Graphics, Inc., 2005.

[27] M. Kerrisk. CAP_SYS_ADMIN: the new root. `http://lwn.net/Articles/486306/`, March 2012.

[28] M. Kerrisk. User namespaces progress. `https://lwn.net/Articles/528078/`, Dec. 2012.

[29] S. Kramer. LINUS IV — An Experiment in Computer Security. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 24–33, 1984.

[30] M. Larabel. A NVIDIA Tegra 2 DRM/KMS Driver Tips Up. `http://www.phoronix.com/vr.php?view=MTA4NjA`, Apr. 2012.

[31] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[32] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Security Symposium*, pages 29–42, 2001.

[33] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Security Symposium*, pages 259–270, 1993.

[34] Microsoft. TCP/IP Raw Sockets. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms740548\%28v=vs.85\%29.aspx`, 2012.

[35] MITRE. Common Vulnerabilities and Exploits Database. `http://cve.mitre.org/`, Feb. 2013.

[36] Summary about POSIX 1.e. `http://wt.tuxomania.net/publications/posix.1e/`, Feb. 1999.

[37] N. Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 257–272, 2002.

[38] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer System. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[39] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, 1999.

[40] I. Shields. Monitor linux file system events with inotify. `http://www.ibm.com/developerworks/linux/library/l-inotify/index.html`, Sep. 2010.

[41] T. Shinagawa and K. Kono. Implementing A Secure Setuid Program. In *Proceedings of the Conference on Parallel and Distributed Computing and Networks*, pages 301–309, 2004.

[42] M. M. Swift, P. Brundrett, C. V. Dyke, P. Garg, A. Hopkins, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control in Windows NT. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 87–96, 2001.

[43] A. Tominaga, O. Nakamura, F. Teraoka, and J. Murai. Problems and Solutions of DHCP - Experiences with DHCP implementation and Operation. `http://www.isoc.org/inet95/proceedings/PAPER/127/html/paper.html`, 1995.

[44] D. Tsafrir, D. D. Silva, and D. Wagner. The murky issue of changing process identity: revising "setuid demystified". *USENIX ;login*, 33(3):55–66, June 2008.

[45] Ubuntu Wiki: Filesystem Capabilities. `https://wiki.ubuntu.com/Security/Features\#Filesystem_Capabilities`, March 2014.

[46] 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 *Proceedings of the USENIX Security Symposium*, pages 3–3, 1996.

[47] R. Wong. A comparison of secure UNIX operating systems. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 322–333, 1990.

[48] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the USENIX Security Symposium*, pages 17–31, 2002.

[49] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM (CACM)*, 17(6):337–345, June 1974.