Linux Security Modules:
General Security Support for the Linux Kernel

Chris Wright and Crispin Cowan [*]
WireX Communications, Inc.

Stephen Smalley [†]
NAI Labs, Network Associates, Inc.

James Morris
Intercode Pty Ltd

Greg Kroah-Hartman [‡]
IBM Linux Technology Center

August 17, 2002

Abstract

The access control mechanisms of existing mainstream operating systems are inadequate to provide strong system security. Enhanced access control mechanisms have failed to win acceptance into mainstream operating systems due in part to a lack of consensus within the security community on the right solution. Since general-purpose operating systems must satisfy a wide range of user requirements, any access control mechanism integrated into such a system must be capable of supporting many different access control models. The Linux Security Modules (LSM) project has developed a lightweight, general purpose, access control framework for the mainstream Linux kernel that enables many different access control models to be implemented as loadable kernel modules. A number of existing enhanced access control implementations, including Linux capabilities, Security-Enhanced Linux (SELinux), and Domain and Type Enforcement (DTE), have already been adapted to use the LSM framework. This paper presents the design and implementation of LSM and discusses the challenges in providing a truly general solution that minimally impacts the Linux kernel.

1 Introduction

The critical role of operating system protection mechanisms in providing system security has been well-understood for over thirty years, yet the access control mechanisms of existing mainstream operating systems are still inadequate to provide strong security [2, 37, 27, 16, 25, 6, 29]. Although many enhanced access control models and frameworks have been proposed and implemented [9, 1, 4, 39, 22, 10, 28, 35], mainstream operating systems typically still lack support for these enhancements. In part, the absence of such enhancements is due to a lack of agreement within the security community on the right general solution.

Like many other general-purpose operating systems, the Linux kernel only provides discretionary access controls and lacks any direct support for enhanced access control mechanisms. However, Linux has long supported dynamically loadable kernel modules, primarily for device drivers, but also for other components such as filesystems. In principle, enhanced access controls could be implemented as Linux kernel modules, permitting many different security models to be supported.

In practice, creating effective security modules is problematic since the kernel does not provide any infrastructure to allow kernel modules to mediate access to kernel objects. As a result, kernel modules typically resort to system call interposition to control kernel operations [17, 19], which has serious limitations as a method for providing access control [39]. Furthermore, these kernel modules often require re-implementing selected kernel functionality [17, 19] or require a patch to the kernel to support the module [10, 3, 14], reducing much of the value of modular composition. Hence,

[*] This work supported in part by DARPA Contract N66001-00-C-8032 (Autonomix)

[†] This work supported by NSA Contract MDA904-01-C-0926 (SELinux)

[‡] This work represents the view of the authors and does not necessarily represent the view of IBM.

many projects have implemented enhanced access control frameworks or models for the Linux kernel as kernel patches [28, 35, 22, 31, 26].

At the Linux Kernel 2.5 Summit, the NSA presented their work on Security-Enhanced Linux (SELinux) [28], an implementation of a flexible access control architecture in the Linux kernel, and emphasized the need for such support in the mainstream Linux kernel. Linus Torvalds appeared to accept that a general access control framework for the Linux kernel is needed, but favored a new infrastructure that would provide the necessary support to kernel modules for implementing security. This approach would avoid the need to choose among the existing competing projects.

In response to Linus' guidance, the Linux Security Modules (LSM) [43, 38] project has developed a lightweight, general purpose, access control framework for the mainstream Linux kernel that enables many different access control models to be implemented as loadable kernel modules. A number of existing enhanced access control implementations, including POSIX.1e capabilities [40], SELinux, and Domain and Type Enforcement (DTE) [22], have already been adapted to use the LSM framework.

The LSM framework meets the goal of enabling many different security models with the same base Linux kernel while minimally impacting the Linux kernel. The generality of LSM permits enhanced access controls to be effectively implemented without requiring kernel patches. LSM also permits the existing security functionality of POSIX.1e capabilities to be cleanly separated from the base kernel. This allows users with specialized needs, such as embedded system developers, to reduce security features to a minimum for performance. It also enables development of POSIX.1e capabilities to proceed with greater independence from the base kernel.

The remainder of this paper is organized as follows. Section 2 elaborates on the problem that LSM seeks to solve. Section 3 presents the LSM design. Section 4 presents the current LSM implementation. Section 5 describes the operational status of LSM, including testing, performance overhead, and modules built for LSM so far. Section 6 describes issues that arose during development, and plans for future work. Section 7 describes related work. Section 8 presents our conclusions.

2 The Problem: Constrained Design Space

The design of LSM was constrained by the practical and technical concerns of both the Linux kernel developers and the various Linux security projects. In email on the topic, Linus Torvalds specified that the security framework must be:

- truly generic, where using a different security model is merely a matter of loading a different kernel module;
- conceptually simple, minimally invasive, and efficient; and
- able to support the existing POSIX.1e capabilities logic as an optional security module.

The various Linux security projects were primarily interested in ensuring that the security framework would be adequate to permit them to re-implement their existing security functionality as a loadable kernel module. The new modular implementation must not cause any significant loss in the security being provided and should have little additional performance overhead.

The core functionality for most of these security projects was access control. However, a few security projects also desired other kinds of security functionality, such as security auditing or virtualized environments. Furthermore, there were significant differences over the range of flexibility for the access controls. Most of the security projects were only interested in further restricting access, i.e. being able to deny accesses that would ordinarily be granted by the existing Linux discretionary access control (DAC) logic. However, a few projects wanted the ability to grant accesses that would ordinarily be denied by the existing DAC logic; some degree of this permissive behavior was needed to support the capabilities logic as a module. Some security projects wanted to migrate the DAC logic into a security module so that they could replace it.

The "LSM problem" is to unify the functional needs of as many security projects as possible, while minimizing the impact on the Linux kernel. The union set of desired features would be highly functional, but also so invasive as to be unacceptable to the mainstream Linux community. Section 3 presents the compromises LSM made to simultaneously balance these conflicting goals.

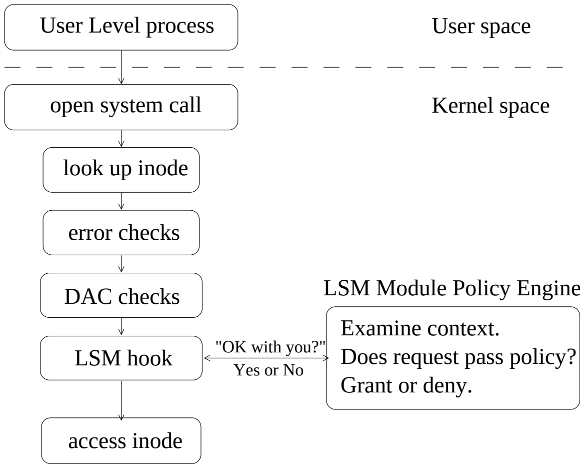

Figure 1: LSM Hook Architecture

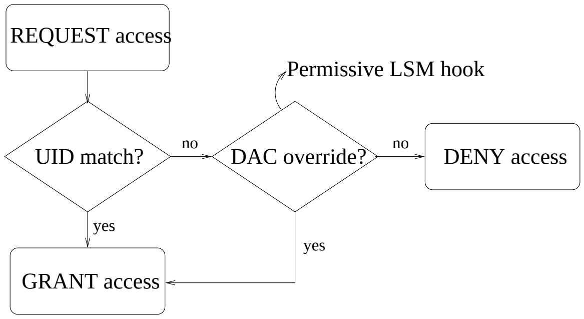

Figure 2: Permissive LSM hook. This hook allows the security policy to override a DAC restriction.

3 LSM Design: Mediate Access to Kernel Objects

The system call interface provides an abstraction for userspace to interact with the kernel, and is a tempting location to mediate access. In fact, no kernel modifications are required to overwrite entries in the system call lookup table, making it trivial to mediate this interface using kernel modules [17, 18]. While this is an attractive feature, mediating the system call interface provides limited value for a general purpose security framework such as LSM [39]. This level of mediation is not race-free, may require code duplication, and may not adequately express the full context needed to make security policy decisions.

The basic abstraction of the LSM interface is to mediate access to *internal* kernel objects. LSM seeks to allow modules to answer the question "May a subject `S` perform a kernel operation `OP` on an internal kernel object `OBJ`?"

LSM allows modules to mediate access to kernel objects by placing *hooks* in the kernel code just ahead of the access, as shown in Figure 1. Just before the kernel *would* have accessed an internal object, a hook makes a call to a function that the LSM module must provide. The module can either let the access occur, or deny access, forcing an error code return.

The LSM framework leverages the kernel's existing mechanisms to translate user supplied data — typically strings, handles or simplified data structures — into internal data structures. This avoids time of check to time of use (TOCTTOU) races [8] and inefficient duplicate look ups. It also allows the LSM framework to directly mediate access to the core kernel data structures. With such an approach, the LSM framework has access to the full kernel context just before the kernel actually performs the requested service. This improves access control granularity.

Given the constrained design space described in Section 2, the LSM project chose to limit the scope of the LSM design to supporting the core access control functionality required by the existing Linux security projects. This limitation enabled the LSM framework to remain conceptually simple and minimally invasive while still meeting the needs of many of the security projects. It also strengthened the justification for adopting the LSM framework into the Linux kernel, since the need for enhanced access controls was more generally accepted by the kernel developers than the need for other kinds of security functionality such as auditing.

A consequence of the "stay simple" design decision is that LSM hooks are primarily *restrictive*: where the kernel was about to grant access, the module may deny access, but when the kernel would deny access, the module is not consulted. This design simplification exists largely because the Linux kernel "short circuits" many decisions early when error conditions are detected. Providing for *authoritative* hooks (where the module can override either decision) would require many more hooks into the Linux kernel.

However, the POSIX.1e capabilities logic requires the ability to grant accesses that would ordinarily be denied at a coarse level of granularity. In order to support this logic as a security module, LSM provides some minimal support for these *permissive* hooks, where the module can grant access the kernel was about to deny. The permissive hooks are typically coupled with a simple DAC check, and allow the module to override the DAC restriction. Figure 2 shows a user access request where a failed user ID check can be overidden by a permissive hook. These hooks are limited to the extent that the kernel already consults the POSIX.1e `capable()` function.

Although LSM was not designed to explicitly support security auditing, some forms of auditing can be supported using the features provided for access control. For example, many of the existing Linux security projects provide support for auditing the access checks performed by their access controls. LSM also enables support for this kind of auditing. Some security auditing can also be supported via existing kernel modules by interposing on system calls, as in the SNARE project [24].

Many security models require binding security attributes to kernel objects. To facilitate this, LSM provides for opaque *security fields* that are attached to various internal kernel objects (detailed in Section 4.1.1). However, the module is completely responsible for managing these fields, including allocation, deallocation, and concurrency control.

Finally, module composition presented a challenge to the LSM design. On the one hand, there clearly is a need to compose some modules with complementary functionality. On the other hand, fully generic security policy composition is known to be intractable [20]. Therefore, LSM *permits* module stacking, but pushes most of the work to the modules themselves. A module that wishes to be stackable must itself export an LSM-like interface, and make calls to subsequently loaded modules when appropriate. The first module loaded has ultimate control over all decisions, determining when to call any other modules and how to combine their results.

4 Implementation

This section describes the implementation of the LSM kernel patch. It begins with an overview of the implementation that describes the types of changes made to the kernel in Section 4.1. Sections 4.2 through 4.7 discuss the specific hooks for the various kernel objects or subsystems.

STRUCTURE	OBJECT
task_struct	Task (Process)
linux_binprm	Program
super_block	Filesystem
inode	Pipe, File, or Socket
file	Open File
sk_buff	Network Buffer (Packet)
net_device	Network Device
kern_ipc_perm	Semaphore, Shared Memory Segment, or Message Queue
msg_msg	Individual Message

Table 1: Kernel data structures modified by the LSM kernel patch and the corresponding abstract objects.

4.1 Implementation Overview

The LSM kernel patch modifies the kernel in five primary ways. First, it adds opaque security fields to certain kernel data structures, described in Section 4.1.1. Second, the patch inserts calls to security hook functions at various points within the kernel code, described in Section 4.1.2. Third, the patch adds a generic security system call, described in Section 4.1.3. Fourth, the patch provides functions to allow kernel modules to register and unregister themselves as security modules, described in Section 4.1.4. Finally, the patch moves most of the capabilities logic into an optional security module, described in Section 4.1.5.

4.1.1 Opaque Security Fields

The opaque security fields are `void*` pointers, which enable security modules to associate security information with kernel objects. Table 1 shows the kernel data structures that are modified by the LSM kernel patch and the corresponding abstract object.

The setting of these security fields and the management of the associated security data is handled by the security modules. LSM merely provides the fields and a set of calls to security hooks that can be implemented by the module to manage the security fields as desired. For most kinds of objects, an `alloc_security` hook and a `free_security` hook are defined that permit the security module to allocate and free security data when the corresponding kernel data structure is allocated and freed. Other hooks are provided to permit the security module to update the security data as necessary, e.g. a `post_lookup` hook that can be used to set security data for an `inode` after a successful lookup operation. It is important to note that LSM does not provide any locking for the security fields; such locking must be performed by the security module.

Since some objects will exist prior to the initialization of a security module, even if the module is built into the kernel, a security module must handle pre-existing objects. Several approaches are possible. The simplest approach is to ignore such objects, treating them as being outside of the control of the module. These objects would then only be controlled by the base Linux access control logic. A second approach is to traverse the kernel data structures during module initialization, setting the security fields for all pre-existing objects at this time. This approach would require great care to ensure that all objects are updated (e.g. an open file might be on a UNIX domain socket awaiting receipt by a process) and to ensure that appropriate locking is performed. A third approach is to test for pre-existing objects on each use and to then set the security field for pre-existing objects when needed.

4.1.2 Calls to Security Hook Functions

As discussed in the previous subsection, LSM provides a set of calls to security hooks to manage the security fields of kernel objects. It also provides a set of calls to security hooks to mediate access to these objects. Both sets of hook functions are called via function pointers in a global `security_ops` table. This structure consists of a collection of substructures that group related hooks based on kernel object or subsystem, as well as some top-level hooks, for system operations. Each hook is defined in terms of kernel objects and parameters, and care has been taken to avoid userspace pointers.

Figure 3 shows the `vfs_mkdir` kernel function after the LSM kernel patch has been applied. This kernel function is used to create new directories. Two calls to security hook functions have been inserted into this function. The first hook call, `security_ops->inode_ops->mkdir`, can be used to control the ability to create new directories. If the hook returns an error status, then the new directory will not be created and the error status will be propagated to the caller. The second hook call, `security_ops->inode_ops->post_mkdir`, can be used to set the security field for the new directory's inode structure. This hook can only update the security module's state; it cannot affect the return status.

```
int vfs_mkdir(struct inode *dir,
  struct dentry *dentry, int mode)
{
   int error;

   down(&dir->i_zombie);
   error = may_create(dir, dentry);
   if (error)
     goto exit_lock;

   error = -EPERM;
   if (!dir->i_op || !dir->i_op->mkdir)
     goto exit_lock;

   mode &= (S_IRWXUGO|S_ISVTX);
   error =
<->  security_ops->inode_ops->mkdir(dir,
                           dentry, mode);
   if (error)
     goto exit_lock;

   DQUOT_INIT(dir);
   lock_kernel();
   error = dir->i_op->mkdir(dir, dentry, mode);
   unlock_kernel();

exit_lock:
   up(&dir->i_zombie);
   if (!error) {
     inode_dir_notify(dir, DN_CREATE);
<->  security_ops->inode_ops->post_mkdir(dir,
                                dentry, mode);
   }
   return error;
}
```

Figure 3: The `vfs_mkdir` kernel function with one security hook call to mediate access and one security hook call to manage the security field. The security hooks are marked by`<->`.

Although LSM also inserts a hook call into the Linux kernel `permission` function, the `permission` hook is insufficient to control file creation operations because it lacks potentially important information, such as the type of operation and the name and mode for the *new* file. Similarly, inserting a hook call into the Linux kernel may_create function would be insufficient since it would still lack precise information about the type of operation and the mode. Hence, a hook was inserted with the same interface as the corresponding inode operation.

An alternative to inserting these two hooks into vfs_mkdir would be to interpose on the dir->i_op->mkdir call. Interposing on internal kernel interfaces would provide equivalent functionality for some of the LSM hooks. However, such interposition would also permit much more general functionality to be implemented via kernel modules. Since kernel modules have historically been allowed to use licenses other than the GPL, an approach based on interposition would likely create political challenges to the acceptance of LSM by the Linux kernel developers.

4.1.3 Security System Call

LSM provides a general security system call that allows security modules to implement new calls for security-aware applications. Although modules can export information and operations via the /proc filesystem or by defining a new pseudo filesystem type, such an approach is inadequate for the needs of some security modules. For example, the SELinux module provides extended forms of a number of existing system calls that permit applications to specify or obtain security information associated with kernel objects and operations.

The security system call is a simple multiplexor fashioned after the existing Linux socketcall system call. It takes the following arguments: (unsigned int id, unsigned int call, unsigned long *args). Since the module defines the implementation of the system call, it can choose to interpret the arguments however it likes. These arguments are intended to be interpreted by the modules as a module identifier, a call identifier, and an argument array. By default, LSM provides a sys_security entrypoint function that simply calls a sys_security hook with the parameters. A security module that does not provide any new calls can define a sys_security hook function that returns -ENOSYS. Most security modules that want to provide new calls can place their call implementations in this hook function.

In some cases, the entrypoint function provided by LSM may be inadequate for a security module. For example, one of the new calls provided by SELinux requires access to the registers on the stack. The SELinux module implements its own entrypoint function to provide such access, and replaces the LSM entrypoint function with this function in the system call table during module initialization.

4.1.4 Registering Security Modules

The LSM framework is initialized during the kernel's boot sequence with a set of dummy hook functions that enforce traditional UNIX superuser semantics. When a security module is loaded, it must register itself with the LSM framework by calling the register_security function. This function sets the global security_ops table to refer to the module's hook function pointers, causing the kernel to call into the security module for access control decisions. The register_security function will not overwrite a previously loaded module. Once a security module is loaded, it becomes a policy decision whether it will allow itself to be unloaded.

If a security module is unloaded, it must unregister with the framework using unregister_security. This simply replaces the hook functions with the defaults so the system will still have some basic means for security. The default hook functions do not use the opaque security fields, so the system's security should not be compromised if the module does a poor job of resetting the opaque fields.

As mentioned in Section 3, general composition of policies is intractable. While arbitrary policy composition gives undefined results, it is possible to develop security modules such that they can compose with defined results. To keep the framework simple, it is aware of only one module, either the default or the registered module – the primary module. A security module may register itself directly with the primary module using the mod_reg_security interface. This registration is controlled by the primary module, so it is a policy decision whether to allow module stacking. With this simple interface, basic module stacking can be supported with no complexity in the framework.

4.1.5 Capabilities

The Linux kernel currently provides support for a subset of POSIX.1e capabilities. One of the requirements for the LSM project was to move this functionality to an optional security module, as mentioned in Section 2. POSIX.1e capabilities provides a mechanism for partitioning traditional superuser privileges and assigning them to particular processes.

By nature, privilege granting is a permissive form of access control, since it grants an access that would ordinarily be denied. Consequently, the LSM framework had to provide a permissive interface with at least the same granularity of the Linux capabilities implementation. LSM retains the existing `capable` interface used within the kernel for performing capability checks, but reduces the `capable` function to a simple wrapper for a LSM hook, allowing any desired logic to be implemented in the security module. This approach allowed LSM to leverage the numerous (more than 500) existing kernel calls to `capable` and to avoid pervasive changes to the kernel. LSM also defines hooks to allow the logic for other forms of capability checking and capability computations to be encapsulated within the security module.

A process capability set, a simple bit vector, is stored in the `task_struct` structure. Because LSM adds an opaque security field to the `task_struct` and hooks to manage the field, it would be possible to move the existing bit vector into the field. Such a change would be logical under the LSM framework but this change has not been implemented in order to ease stacking with other modules. One of the difficulties of stacking security modules in the LSM framework is the need to share the opaque security fields. Many security modules will want to stack with the capabilities module, because the capabilities logic has been integrated into the mainstream kernel for some time and is relied upon by some applications such as `named` and `sendmail`. Leaving the capability bit vector in the `task_struct` eases this composition at the cost of wasted space for modules that don't need to use it.

The Linux kernel support for capabilities also includes two system call calls: `capset` and `capget`. To remain compatible with existing applications, these system calls are retained by LSM but the core capabilities logic for these functions has been replaced by calls to LSM hooks. Ultimately, these calls should be reimplemented via the `security` system call. This change should have little impact on applications since the portable interface for capabilities is through the `libcap` library rather than direct use of these calls.

The LSM project has developed a capabilities security module and migrated much of the core capabilities logic into it; however, the kernel still shows vestiges of the pre-existing Linux capabilities. Moving the bit vector from the `task_struct` proper to the opaque security field and relocating the system call interface are the only major steps left to making the capability module completely standalone.

4.2 Task Hooks

LSM provides a set of task hooks that enable security modules to manage process security information and to control process operations. Modules can maintain process security information using the security field of the `task_struct` structure. Task hooks provide control over inter-process operations, such as `kill`, as well as control over privileged operations on the current process, such as `setuid`. The task hooks also provide fine-grained control over resource management operations such as `setrlimit` and `nice`.

4.3 Program Loading Hooks

Many security modules, including Linux capabilities, DTE, SELinux, and SubDomain require the ability to perform changes in privilege when a new program is executed. Consequently, LSM provides a set of program-loading hooks that are called at critical points during the processing of an `execve` operation. The security field of the `linux_binprm` structure permits modules to maintain security information during program loading. One hook is provided to permit security modules to initialize this security information and to perform access control prior to loading the program, and a second hook is provided to permit modules to update the task security information after the new program has been successfully loaded. These hooks can also be used to control inheritance of state across program executions, for example, revalidating open file descriptors.

4.4 IPC Hooks

Security modules can manage security information and perform access control for System V IPC using the LSM IPC hooks. The IPC object data structures share a common substructure, `kern_ipc_perm`, and only a pointer to this substructure is passed to the existing `ipcperms` function for checking permissions. Hence, LSM adds a security field to this shared substructure. To support security information for individual messages, LSM also adds a security field to the `msg_msg` structure.

LSM inserts a hook into the existing ipcperms function so that a security module can perform a check for each existing Linux IPC permission check. However, since these checks are not sufficient for some security modules, LSM also inserts hooks into the individual IPC operations. These hooks provide more detailed information about the type of operation and the specific arguments. They also support fine-grained control over individual messages sent via System V message queues.

4.5 Filesystem Hooks

For file operations, three sets of hooks were defined: filesystem hooks, inode hooks, and file hooks. LSM adds a security field to each of the associated kernel data structures: super_block, inode, and file. The filesystem hooks enable security modules to control operations such as mounting and statfs. LSM leverages the existing permission function by inserting an inode hook into it, but LSM also defines a number of other inode hooks to provide finer-grained control over individual inode operations. Some of the file hooks allow security modules to perform additional checking on file operations such as read and write, for example, to revalidate permissions on use to support privilege bracketing or dynamic policy changes. A hook is also provided to allow security modules to control receipt of open file descriptors via socket IPC. Other file hooks provide finer-grained control over operations such as fcntl and ioctl.

An alternative to placing security fields in the inode and super_block structures would have been to place them in the dentry and vfsmount structures. The inode and super_block structures correspond to the actual objects and are independent of names and namespaces. The dentry and vfsmount structures contain a reference to the corresponding inode or super_block, and are associated with a particular name or namespace. Using the first pair of structures avoids object aliasing issues. The use of these structures also provides more coverage of kernel objects, since these structures also represent non-file objects such as pipes and sockets. These data structures are also readily available at any point in the filesystem code, whereas the second set of structures is often unavailable.

4.6 Network Hooks

Application layer access to networking is mediated using a set of socket hooks. These hooks, which include the interposition of all socket system calls, provide coarse mediation coverage of all socket-based protocols. Since active user sockets have an associated inode structure, a separate security field was not added to the socket structure or to the lower-level sock structure. As the socket hooks allow general mediation of network traffic in relation to processes, LSM significantly expands the kernel's network access control framework (which is already handled at the network layer by Netfilter [34]). For example, the sock_rcv_skb hook allows an inbound packet to be mediated in terms of its destination application, prior to being queued at the associated userspace socket.

Additional finer-grained hooks have been implemented for the IPv4, UNIX domain, and Netlink protocols, which were considered essential for the implementation of a minimally useful system. Similar hooks for other protocols may be implemented at a later stage.

Network data traverses the stack in packets encapsulated by an sk_buff (socket buffer) structure. LSM adds a security field to the sk_buff structure, so that security state may be managed across network layers on a per-packet basis. A set of sk_buff hooks is provided for lifecycle management of this security field.

Hardware and software network devices are encapsulated by a net_device structure. A security field was added to this structure so that security state can be maintained on a per-device basis.

Coverage of low level network support components, such as routing tables and traffic classifiers is somewhat limited due to the invasiveness of the code which would be required to implement consistent fine-grained hooks. Access to these objects can be mediated at higher levels (for example, using ioctl), although granularity may be reduced by TOCTTOU issues.

4.7 Other Hooks

LSM provides two additional sets of hooks: module hooks and a set of top-level *system* hooks. Module hooks can be used to control the kernel operations that create, initialize, and delete kernel modules. System hooks can be used to control system operations, such as setting the system hostname, accessing I/O ports, and configuring process accounting. The existing Linux kernel provides some control over many of these operations using the capability checks, but those checks only provide coarse-grained distinctions among different operations and do not provide any argument information.

5 Testing and Functionality

Section 5.1 surveys modules that have been created for LSM so far. Section 5.2 describes our performance testing of LSM. While we have tested LSM kernels by booting and running them, *we* have not engaged in systematic testing. However, other members of the LSM community [43] have developed systematic LSM correctness testing procedures [13].

5.1 Modules

LSM provides only the mechanism to enforce enhanced access control policies. Thus, it is the LSM modules that implement a specific policy and are critical in proving the functionality of the framework. Below are briefly described a few of these LSM modules:

- **SELinux** A Linux implementation of the Flask [39] flexible access control architecture and an example security server that supports Type Enforcement, Role-Based Access Control, and optionally Multi-Level Security. SELinux was originally implemented as a kernel patch [28] and was then reimplemented as a security module that uses LSM. SELinux can be used to confine processes to least privilege, to protect the integrity and confidentiality of processes and data, and to support application security needs. The generality and comprehensiveness of SELinux helped to drive the requirements for LSM.
- **DTE Linux** An implementation of Domain and Type Enforcement [4, 5] developed for Linux [22]. Like SELinux, DTE Linux was originally implemented as a kernel patch and was then adapted to LSM. With this module loaded, types can be assigned to objects and domains to processes. The DTE policy restricts access between domains and from domains to types. The DTE Linux project also provided useful input into the design and implementation of LSM.
- **LSM port of Openwall kernel patch** The Openwall kernel patch [12] provides a collection of security features to protect a system from common attacks, e.g. buffer overflows and temp file races. A module is under development that supports a subset of the Openwall patch. For example, with this module loaded a victim program will not be allowed to follow malicious symlinks.
- **POSIX.1e capabilities** The POSIX.1e capabilities [40] logic was already present in the Linux kernel, but the LSM kernel patch cleanly separates this logic into a security module. This change allows users who do not need this functionality to omit it from their kernels and it allows the development of the capabilities logic to proceed with greater independence from the main kernel.

5.2 Performance Overhead

The LSM kernel used for the tests included the POSIX.1e capabilities security module in order to provide a fair comparison between an unmodified Linux kernel with built-in capabilities support and a LSM kernel with a capabilities module.

5.2.1 Microbenchmark: LMBench

We used LMBench [30] for microbenchmarking. LMBench was developed specifically to measure the performance of core kernel system calls and facilities, such as file access, context switching, and memory movement. LMBench has been particularly effective at establishing and maintaining excellent performance in these core facilities in the Linux kernel.

We compared a standard Linux 2.5.2 kernel against a 2.5.2 kernel with the LSM patch applied, run on a 4-processor 700 MHz Pentium Xeon computer with 1 GB of RAM and an ultra-wide SCSI disk, with the results shown in Table 2. In most cases, the performance penalty is in the experimental noise range. In some cases, the LSM kernel's performance actually exceeded the standard kernel, which we attribute to experimental error (typically cache collision anomalies [23]).

The worst case overhead was 6.2% for `stat()`, 6.6% for `open/close`, and 7.2% for file delete. These results are to be expected because of the relatively small amount of work done in each call compared to the work of checking for LSM mediation. Similar results for a 1-processor 433 MHz Pentium 3 with 128 MB of RAM and an IDE disk are shown in Table 3. One should also bear in mind that these are microbenchmark figures; for comprehensive application-level impact, see

Process tests, times in μseconds, smaller is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
null call	0.50	0.49	-2.0%
null I/O	0.82	0.84	2.4%
stat	5.95	6.34	6.2%
open/close	7.59	8.13	6.6%
sig inst	1.26	1.25	-0.8%
sig handl	3.86	3.92	1.5%
fork proc	155	157	1.3%
exec proc	917	924	0.8%
sh proc	3834	3881	1.2%

File and VM system latencies in μseconds, smaller is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
0K file create	40	40	0%
0K file delete	6.737	7.258	7.2%
10K file create	104	105	0.1%
10K file delete	24	25	4.0%
mmap latench	2641	2699	2.1%
prot fault	1.014	0.985	-2.9%
page fault	3.00	3.00	0%

Local communication bandwidth in MB/s, larger is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
TCP bandwidth	216	219	-1.4%
file reread	309	311	-0.6%
mmap reread	368	372	-1.1%
bcopy (libc)	191	195	-2.1%
bcopy (hand)	149	158	-6.0%
mem read	368	371	-0.8%
mem write	197	201	-2.0%

Table 2: LMBench Microbenchmarks, 4 processor machine

Process tests, times in μseconds, smaller is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
null call	0.70	0.72	2.7%
null I/O	1.06	1.08	1.9%
stat	45.9	46.8	1.9%
open/close	47.5	48.6	2.3%
sig inst	1.95	1.94	-0.5%
sig handl	8.39	8.53	1.6%
fork proc	455	459	0.9%
exec proc	2024	2010	-0.7%
sh proc	10000	9980	-0.2%

File and VM system latencies in μseconds, smaller is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
0K file create	111.6	113.2	1.4%
0K file delete	52.1	53.7	3.0%
10K file create	235.6	237.0	0.6%
10K file delete	71.1	72.2	1.5%
mmap latench	469.0	437.0	-7.3%
prot fault	1.497	1.520	1.5%
page fault	4.75	4.25	-11.8%

Local communication bandwidth in MB/s, larger is better:

test type	2.5.2	2.5.2-lsm	%overhead with LSM
TCP bandwidth	anomaly	anomaly	%
file reread	149.0	148.8	0.1%
mmap reread	247.4	247.4	0%
bcopy (libc)	110.1	110.8	-0.6%
bcopy (hand)	109.3	109.8	-0.5%
mem read	247	247	0%
mem write	153.2	153.6	-0.2%

Table 3: LMBench Microbenchmarks, 1 processor machine

Section 5.2.2.

5.2.2 Macrobenchmark: Kernel Compilation

Our macrobenchmark is the widely used kernel compilation benchmark, measuring the time to build the Linux kernel. We ran this test on both a single processor machine (433 MHz Pentium 3, 128 MB RAM, IDE disk), and a 4-processor SMP (four 700 MHh Xeon processors, 1 GB RAM, ultra wide SCSI disk).

The single processor test executed the command `time make -j2 bzImage` and the 4-processor test executed the command `time make -j8 bzImage`, with the results shown in Table 4. In all cases, the LSM kernel performed slightly better than the non-LSM kernel. As in our microbenchmark experiments in Section 5.2.1, we regard this as experimental error, and claim only that LSM overhead at the application level is not significant.

6 Discussion

Given that LSM set out to satisfy the needs of a collection of other independent projects, it is understandable that the result produced some emergent properties.

Machine type	2.5.2	2.5.2-lsm	%overhead with LSM
4 CPUs	98.38	98.26	-0.1%
1 CPU	659.36	649.78	-1.5%

Table 4: Linux Kernel Build Macrobenchmarks, time in seconds

Many security models require some way to associate security attributes to system objects. Thus LSM attaches security fields to many internal kernel objects so that modules may attach and later reference the security attributes associated with those objects.

It is also desirable to *persistently* bind security attributes to files. To do so seamlessly requires *extended attribute* file system support, which enables security attributes to be bound to files on disk. However, supporting extended attributes is a complex issue, requiring both support for extended attributes in the filesystem [21], and support for extended attributes in the Linux kernel's VFS layer. Rather than make LSM dependent on non-existent kernel features, we plan to add full extended attribute support to LSM when extended attributes are added to the Linux kernel. Until then, modules that need persistent extended attributes can resort to meta-files [42, 28].

In attempting to provide a pluggable interface for security enhancements, it is tempting to consider *completely* modularizing all security policy decisions, i.e. move *all* kernel logic concerning access control out of the kernel and into a default module. This approach has significant benefits beyond simple modular consistency: in particular, it would make it much easier to provide *authoritative* hooks instead of *restrictive* hooks, which in turn would enable a broader variety of modules (see Section 3).

However, we chose *not* to modularize all security decisions, for pragmatic reasons. Current Linux access control decisions are not well isolated in the kernel; they are mingled with other error checking and transformation logic. Thus a patch to the Linux kernel to remove all access control logic would be highly invasive. Implementing such a change would almost certainly entail security bugs, which would not be an auspicious way to introduce LSM to the greater Linux community.

Therefore, we deferred the complete modularization of all access control logic. The current LSM implements much less invasive restrictive hooks, providing a minimally invasive patch for initial introduction into the Linux community. Once LSM is well established, we may revisit this decision, and propose a more radical modularization architecture.

Finally, in designing the LSM interface, we were distinctly aware that LSM constitutes an API, and thus must present a logically consistent view to the programmer. The LSM interface constitutes not only the set of hooks needed by the modules we intended to support, but also the logical extension of such hooks, such that the interface is regular. Where possible, special cases were generalized so that they were no longer special.

7 Related Work

Section 7.1 describes the general area of extensible kernels in the LSM context, and Section 7.2 describes work specifically related to generic access control frameworks.

7.1 Extensible Kernel Research

There has been a lot of operating systems research in the last 20 years on extensible systems. Following the basic idea of microkernels (which sought to componentize most everything in the kernel) came extensive efforts to build more monolithic kernels that could be extended in various ways:

- **Exokernel** was really just a logical extension of the microkernel concept [15]. The base kernel provided no abstraction of physical devices, leaving that to applications that needed the devices.

- **SPIN** allowed modules to be loaded into the kernel, while providing for a variety of safety properties [7]. Modules were to be written in Modula-3 [33], which imposed strong type checking, thus preventing the module from mis-behaving outside of its own data structures. SPIN "spindles" also were subject to time constraints, so they could not seize the CPU. Abstractly, spindles would register to "extend" or "specialize" kernel events, and would be added to an event handling chain, rather similar to the way interrupts are commonly handled.

- **SCOUT** was designed to facilitate continuous flows of information (e.g. audio or video streams), and allowed CODEC stages to be composed into pipelines (or graphs) of appropriate components [32].

- **Synthetix** sought to allow applications to *specialize* the operating system to their transient needs [36]. “Specialization” meant optimization with respect to “quasi-invariants”: properties that hold true for a while, but eventually become false. In some cases, quasi-invariants were inferred from application behavior, such as a process opening a file, resulting in a specialized `read()` system call optimized for the particular process and file. In other cases, quasi-invariants were specified to the kernel using a declarative language [11, 41].

All of these extension facilities provided some form of safety, to limit the potential damage that an extension could impose on the rest of the system. Such safety properties, for example, might allow a multimedia application to extend the kernel to support better quality of service, while limiting the multimedia extension so that it does not accidentally corrupt the operating system. The need for such safety in kernel extensions is anecdotally confirmed by the phenomena of unstable Microsoft Windows systems, which are allegedly made unstable in part due to bad 3rd party device drivers, which run in kernel space.

In contrast, LSM imposes no restrictions on modules, which are (normally) written in C and have full, untyped access to the kernel’s address space. The only “restriction” is that hooks are mostly of the “restrictive” form, making it somewhat more difficult to “oops” and grant access when it should have been denied. Rather, LSM depends primarily on programmer skill (modules need to be written with the dilligence of kernel code) and root authority (only root may load a module).

It should be noted that LSM can get away with this weak module safety policy *precisely* because LSM modules are intended to enforce security policy. Unlike more generic kernel extensions such as QoS, the system is entirely at the mercy of the security policy. An administrator who permits an LSM module to be loaded has already made the decision to trust the module providers to be both well-intentioned and skilled at programming, as bugs in a security policy engine can have catastrophic consequences. Further sanity checks on LSM modules are superfluous.

It should also be noted that this is the traditional view of Linux modules: that loading modules into the kernel is privileged for a reason, and that care should be taken in the writing and selection of kernel modules. LSM module developers are cautioned to be especially dilligent in creating modules. Not only do LSM modules run with the full authority of all kernel code, but they are espcially trusted to enforce security policy correctly. Third party review of LSM modules’ source code is recommended.

Finally, we note that LSM is much less intrusive to the Linux kernel than the other large modular interface: VFS (Virtual Filesystem). The need for support for multiple filesystems in Linux was recognized long ago, and thus a rich infrastructure was built. The VFS layer of the kernel abstracts the features of most filesystems, so that other parts of the kernel can access the filesystem without what knowing what kind of filesystem is in use. Anecdotally, the VFS layer is reported to be a nest of function pointers that was very difficult to debug. This difficulty may explain, in part, why the Linux community would like the LSM interface to be as minimally intrusive as possible.

7.2 General Access Control Frameworks

The challenge of providing a highly general access control framework has been previously explored in the Generalized Framework for Access Control (GFAC) [1] and the Flask architecture [39]. These two architectures have been implemented as patches for the Linux kernel by the RSBAC [35] and the SELinux [28] projects. The Medusa [31] project has developed its own general access control framework [44] and implemented it in Linux. Domain and Type Enforcement (DTE) [4] provides support for configurable security policies, and has also been implemented in Linux [22].

Like these prior projects, LSM seeks to provide general support for access control in the Linux kernel. However, the goals for LSM differ from these projects, yielding corresponding differences in the LSM framework. In particular, the emphasis on minimal impact to the base Linux kernel, the separation of the capabilities logic, and the need to support security functionality as kernel modules distinguish LSM from these prior projects.

Additionally, since LSM seeks to support a broad range of existing Linux security projects, it cannot impose a particular access control architecture such as Flask or the GFAC or a particular model such as DTE. In order to provide the greatest flexibility, LSM simply exposes the kernel abstractions and operations to the security modules, allowing the individual modules to implement their desired architecture or model. Similarly, since the various projects use significantly different approaches for associating security attributes with files, LSM defers file labeling support entirely to the module. For systems like SELinux or RSBAC, this approach introduces a new level of indirection, so that even the general access control architecture and the file labeling support would be encapsulated within the module rather than being directly integrated into the kernel.

8 Conclusions

The Linux kernel supports the classical UNIX security policies of mode bits, and a partial implementation of the draft POSIX.1e "capabilities" standard, which in many cases is not adequate. The combination of open source code and broad popularity has made Linux a popular target for enhanced security projects. While this *works*, in that many powerful security enhancements are available, it presents a significant barrier to entry for users who are unable or unwilling to deploy custom kernels.

The Linux Security Modules (LSM) project exists to ease this barrier to entry by providing a standard loadable module interface for security enhancements. We presented the motivation, design, and implementation of the LSM interface. LSM provides an interface that is rich enough to enable a wide variety of security modules, while imposing minimal disturbance to the Linux source code, and minimal performance overhead on the Linux kernel. Several robust security modules are already available for LSM.

LSM is currently implemented as a patch to the standard Linux kernel. A patch is being maintained for the latest versions of the 2.4 stable series and the 2.5 development series. The goal of the LSM project is for the patch to be adopted into the standard Linux kernel as part of the 2.5 development series, and eventually into most Linux distributions.

References

[1] Marshall D. Abrams, Leonard J. LaPadula, Kenneth W. Eggers, and Ingrid M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, pages 135–143, October 1990.

[2] J. Anderson. Computer Security Technology Planning Study. Report Technical Report ESD-TR-73-51, Air Force Elect. Systems Div., October 1972.

[3] Argus Systems. PitBull LX. `http://www.argus-systems.com/product/white_paper/lx`.

[4] L. Badger, D.F. Sterne, and et al. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.

[5] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the USENIX Security Conference*, 1995.

[6] D. Baker. Fortresses built upon sand. In *Proceedings of the New Security Paradigms Workshop*, 1996.

[7] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[8] M. Bishop and M. Digler. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996. Also available at `http://olympus.cs.ucdavis.edu/~bishop/scriv/index.html`.

[9] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.

[10] Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, December 2000.

[11] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization Classes: An Object Framework for Specialization. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, Seattle, WA, October 27-28 1996.

[12] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[13] Antony Edwards, Trent R. Jaeger, and Xiaolan Zhang. Verifying Authorization Hook Placement for the Linux Security Modules Framework. Report RC22254, IBM T.J. Watson Research Center, December 2001. `http://domino.watson.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b3600%66f0d4/fd3bffacfd2bbd9385256b30005ec7ee?OpenDocument`.

[14] Nigel Edwards, Joubert Berger, and Tse Houng Choo. A Secure Linux Platform. In *Proceedings of the 5th Annual Linux Showcase and Conference*, November 2001.

[15] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[16] M. Abrams et al. *Information Security: An Integrated Collection of Essays*. IEEE Comp., 1995.

[17] Tim Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[18] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[19] Timothy Fraser. LOMAC: MAC You Can Live With. In *Proceedings of the FREENIX Track, USENIX Annual Technical Conference*, Boston, MA, June 2001.

[20] Virgil D. Gligor, Serban I Gavrila, and David Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[21] Andreas Grunbacher. Extended Attributes and Access Control Lists for Linux. World-wide web page available at `http://acl.bestbits.at/`, December 2001.

[22] Serge Hallyn and Phil Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[23] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. *Operating Systems Review*, 24(4):896–908, October 1992. Also published as OGI technical report CSE-92-010, ftp://cse.ogi.edu/pub/tech-reports/1992/92-010.ps.gz.

[24] SNARE. World-wide web page available at `http://intersectalliance.com/projects/Snare/`.

[25] Jay Lepreau, Bryan Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proceedings of the ACM SIGOPS European Workshop*, pages 133–140, September 1996.

[26] Linux Intrusion Detection System. World-wide web page available at `http://www.lids.org`.

[27] T. Linden. Operating System Structures to Support Security and Reliable Software. *ACM Computing Surveys*, 8(4), December 1976.

[28] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[29] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, October 1998.

[30] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996. `http://www.bitmover.com/lmbench/`.

[31] Medusa. World-wide web page available at `http://medusa.fornax.sk`.

[32] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–168, October 1996. `http://www.cs.arizona.edu/scout/Papers/osdi96/`.

[33] Greg Nelson. *System Programming in Modula-3*. Prentice Hall, 1991.

[34] Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4, 1999. `http://www.netfilter.org/`.

[35] Amon Ott. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In *Proceedings of the 8th International Linux Kongress*, November 2001.

[36] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[37] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[38] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. `http://lsm.immunix.org/`, September 2001.

[39] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.

[40] Winfried Trumper. Summary about POSIX.1e. `http://wt.xpilot.org/publications/posix.1e`, July 1999.

[41] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, Atlanta, GA, October 1997.

[42] Robert N.M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[43] WireX Communications. Linux Security Module. `http://lsm.immunix.org/`, April 2001.

[44] Marek Zelem and Milan Pikula. ZP Security Framework. `http://medusa.fornax.sk/English/medusa-paper.ps`.