Linux Security Module Framework

Chris Wright and Crispin Cowan *
WireX Communications, Inc.
chris@wirex.com, crispin@wirex.com

Stephen Smalley †
NAI Labs, Network Associates, Inc.
sds@tislabs.com

James Morris
Intercode Pty Ltd
jmorris@intercode.com.au

Greg Kroah-Hartman ‡
IBM Linux Technology Center
gregkh@us.ibm.com

Abstract

Computer security is a chronic and growing problem, even for Linux, as evidenced by the seemingly endless stream of software security vulnerabilities. Security research has produced numerous access control mechanisms that help improve system security; however, there is little consensus on the best solution. Many powerful security systems have been implemented as research prototypes or highly specialized products, leaving systems operators with a difficult challenge: how to utilize these advanced features, without having to throw away their existing systems?

The Linux Security Modules (LSM) project addresses this problem by providing the Linux kernel with a general purpose framework for access control. LSM enables loading enhanced security policies as kernel modules. By providing Linux with a standard API for policy enforcement modules, the LSM project hopes to enable widespread deployment of security hardened systems. This paper presents the design and implementation of the LSM framework, a discussion of performance and security impact on the kernel, and a brief overview of existing security modules.

*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. But that sentence did.

1 Introduction

Security is a chronic and growing problem: as more systems (and more money) go on line, the motivation to attack rises. Linux is not immune to this threat: the "many eyes make shallow bugs" argument [25] not withstanding, Linux systems do experience a large number of software vulnerabilities.

An important way to mitigate software vulnerabilities is through effective use of access controls. Discretionary access controls (`root`, user-IDs and mode bits) are adequate for user management of their own privacy, but are not sufficient to protect systems from attack. Extensive research in non-discretionary access control models has been done for over thirty years [2, 26, 18, 10, 16, 5, 20] but there has been no real consensus on which is the *one true* access control model. Because of this lack of consensus, there are many *patches* to the Linux kernel that provide enhanced access controls [7, 11, 12, 14, 17, 19, 24, 20, 32] but none of them are a *standard* part of the Linux kernel.

The Linux Security Modules (LSM) [30, 27, 31] project seeks to solve this Tower of Babel [1] quandary by providing a general-purpose framework for security policy modules. This allows many different access control models to be implemented as loadable kernel modules, enabling multiple threads of security policy engine development to proceed independently of the main Linux kernel. A number of existing enhanced access control implementations, including POSIX.1e capabilities [29], SELinux, Domain and Type Enforcement (DTE) [14] and Linux Intrusion Detection System (LIDS) [17] have already been adapted to use the LSM framework.

The remainder of this paper is organized as follows. Section 2 presents the LSM design and implementation. Section 3 gives a detailed look at the LSM interface. Section 4 describes the impact LSM has on performance and security, including a look at some projects that have been ported to LSM so far. Section 5 presents our conclusions.

2 Design and Implementation

At the 2001 Linux Kernel Summit, the NSA presented their work on Security-Enhanced Linux (SELinux) [19], an implementation of a flexible access control architecture in the Linux kernel. Linus Torvalds appeared to accept that a general access control framework for the Linux kernel is needed. However, given the many Linux kernel security projects, and Linus' lack of expertise in sophisticated security policy, he preferred an approach that allowed security models to be implemented as loadable kernel modules. In fact, Linus' response provided the seeds of the LSM design. The LSM 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.

To achieve these goals while remaining agnostic with respect to styles of access control mediation, LSM takes the approach of mediating access to the kernel's internal objects: tasks, inodes, open files, etc., as shown in Figure 1. User processes execute system calls, which first traverse the Linux kernel's existing logic for finding and allocating resources, performing error checking, and passing the classical UNIX discretionary access controls. Just before the kernel *attempts to* access the internal object, an LSM **hook** makes an out-call to the module posing the question, "Is this access ok with you?" The module processes this policy question and returns either "yes" or "no."

One might ask why LSM chose this approach rather than *system call interposition* (mediating system calls as they enter the kernel) or *device mediation* (mediating at access to physical devices).[1] The reason is that information critical to sound security policy decisions is not available at those points. At the system call interface, userspace data, such as a path name, has yet to be translated to the kernel object it represents, such as an inode. Thus, system call interposition is both inefficient and prone to time-of-check-to-time-of-use (TOCTTOU) races [28, 6]. At the device interface, some other critical information (such as the path name of the file to be accessed) has been thrown away. In between is where the full context of an access request can be seen, and where a fully informed access control decision can be made.

A subtle implication of the LSM architecture is that access control decisions are *restrictive*[2]: the module can really only say "no" [31]. Functional errors and classical security checks can result in an access request being denied before it is ever presented to the LSM module. This is the opposite of the way mandatory access control systems are normally implemented. This design choice limits the flexibility of the LSM framework, but substantially simplifies the impact of the LSM on the Linux kernel. To do otherwise would have required implementing many instances of the *same* hook throughout the kernel, to ensure that the module is consulted at *every* place where a system call could "error out."

Composition of LSM modules is another problematic issue. On the one hand, security policies do not compose in the *general case* because some policies may explicitly conflict [13]. On the other hand, it is clearly desirable to compose some combinations of security policies. Here, LSM effectively punts to the module writer: to be able to "stack" modules, the first module loaded must also export an LSM interface to subsequent LSM modules to be loaded. The first module is then responsible for composing the access control decisions that it gets back from secondary modules.

[1] The glib answer is that the Linux kernel already provides those features and there would be nothing for us to do :-)

[2] Caveat: the `capable()` hook, which is needed to support POSIX.1e capabilities, can override DAC checks, see Section 3.8.

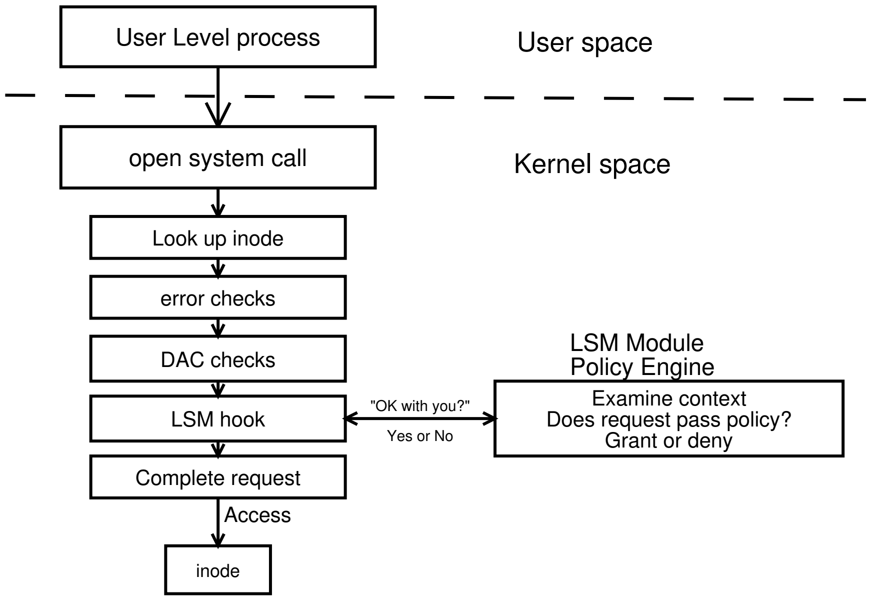

Figure 1: LSM Hook Architecture

3 LSM Interface

Having discussed the high-level design philosophies of LSM in Section 2, we now turn to the implementation of the LSM interface. At the core, the LSM interface is a large table of functions, which by default are populated with calls that implement the traditional superuser DAC policy. The module writers are then responsible for providing implementations of the functions that they care about. This section provides a detailed analysis of those functions.[3] Section 3.1 shows how to register a security module. Sections 3.2 through 3.8 are organized by kernel object and discuss the LSM interface available to mediate access to each object.

3.1 Policy Registration

The LSM interface is implemented as a structure of callback methods, `security_ops`. A security module is responsible for implementing the callbacks according to the security policy it is enforcing. At boot time the `security_ops` structure is initialized with default callbacks, which implement traditional superuser semantics.

The security module can be built as a dynamically loadable module or statically linked into the kernel. It is initialized either at module load time for dynamically loaded modules or during `do_initcalls()` for statically linked modules. During this initialization, the security module must register its callbacks with the LSM framework by calling `register_security()`. A module should call `unregister_security()` when it is unloaded to return the `security_ops` structure to its default superuser policy.

The LSM framework is aware of only one primary security policy at any time. Once a security policy is registered with the LSM framework, subsequent attempts to register new security policies will fail. In some cases it is appropriate to compose security policies, as noted in Section 2. LSM allows modules to stack with each other, however, the framework remains aware of only a single `security_ops` structure. In order to register additional security policies, the subsequent modules register with the primary module using `mod_reg_security()`. This allows the LSM framework to remain simple, pushing the policy which defines composition into the primary security module.

[3] However, it is not a programmer's guide.

3.2 Task Hooks

The task_struct structure is the kernel object representing kernel schedulable tasks. It contains basic task information such as user and group ID, resource limits, and scheduling policies and priorities. LSM provides a group of task hooks, task_security_ops, that mediate a task's access to this basic task information. Interprocess signalling is mediated by the LSM task hooks to monitor tasks' abilities to send and receive signals. LSM adds a security field to the task_struct to allow security policies to label a task with a policy specific security label.

The LSM task hooks have full task life-cycle coverage. The create() task hook is called, verifying that a task can spawn children. If this is successful, a new task is created and the alloc_security() task hook is used to manage the new task's security field. When a task exits, the kill() task hook is consulted to verify that the task can signal its parent. Similarly, the wait() task hook is called in the parent task context, verifying the parent task can receive the child's signal. And finally, the task's security field is released by the free_security() task hook.

During the life of a task it may attempt to change some of its basic task information. For example a task may call setuid(2). This is, of course, managed by LSM with a corresponding setuid() task hook. If this is successful the kernel updates the task's user identity and then notifies the policy module via the post_setuid() task hook. The notification allows the module to update state and, for example, update the task's security field.

To avoid leaking potentially sensitive task information, LSM mediates the ability to query another task's state. So, for example, a query for the process group ID or the scheduler policy of an arbitrary task is protected by the getpgid() or getscheduler() task hooks respectively.

3.3 Program Loading Hooks

The linux_binprm structure represents a new program being loaded during an execve(2). LSM provides a set of program loading hooks, binprm_security_ops, to manage the process of loading new programs. Many security models, including Linux capabilities, require the ability to change privileges when a new program is executed. Consequently, these LSM hooks are called at critical points during program loading to verify a task's ability to load a new program and update the task's security field.

LSM adds a security field to the linux_binprm structure. At the beginning of an execve(2) after the new program file is opened, the alloc_security() program loading hook is called to allocate the security field. The set_security() hook is used to save security information in the linux_binprm security field. This hook may be called multiple times during a single execve(2) to accommodate interpreters. Either of these program loading hooks can be used to deny program execution.

In the final stages of program loading, the compute_creds() program loading hook is called to set the new security attributes of a task being transformed by execve(2). Typically, this hook will calculate the task's new credentials based on both its old credentials and the security information stored in the linux_binprm security field. Once the new program is loaded, the kernel releases the linux_binprm security field by calling the free_security() program loading hook.

3.4 File System Hooks

The VFS layer defines three primary objects which encapsulate the interface that low level filesystems are developed against: the super_block, the inode and the file. Each of these objects contains a set of operations that define the interface between the VFS and the actual filesystem. This interface is a perfect place for LSM to mediate filesystem access. The LSM filesystem hooks are described in Sections 3.4.1 through 3.4.3.

3.4.1 Super Block Hooks

The kernel's super_block structure represents a filesystem. This structure is used when mounting and unmounting a filesystem or obtaining filesystem statistics, for example. The super block hooks, super_block_security_ops, mediate the various actions that can be taken on a super_block. As a simple example, the statfs() super block hook checks permission when a task attempts to obtain a file system's statistics.

When mounting a filesystem, the kernel first validates the request by calling the `mount()` super block hook. Assuming success, a new `super_block` is created[4] regardless of whether it is backed by a block device or by an anonymous device. The kernel then allocates space for a security field in the new `super_block` by calling the `alloc_security()` super block hook. Next, when the `super_block` is to be added to the global tree, the `check_sb()` super block hook is called to verify that the filesystem can indeed be mounted at the point in the tree that is being requested. If this is successful, a `post_addmount()` hook is invoked to synchronize the security module's state.

The super block hook `umount()` is called to check permission when unmounting a filesystem. If successful, the `umount_close()` hook is used to synchronize state and, for example, close any files in the filesystem that are held open by the security module. Once the `super_block` is no longer referenced, it will be deleted, and the `free_security()` hook will free the security field.

3.4.2 Inode Hooks

The kernel's `inode` structure represents a basic filesystem object, e.g., a file, directory, or symlink. The LSM inode hooks mediate access to this fundamental kernel structure. A well defined set of operations, `inode_operations`, describe the actions that can be taken on an inode — `create()`, `unlink()`, `lookup()`, `mknod()`, `rename()`, and so on. This encapsulation defines a nice interface for LSM to mediate access to the `inode` object. In addition, LSM adds a security field to the `inode` structure and corresponding inode hooks to manage security labelling.

The kernel's `inode` cache is populated by either file lookup operations or filesystem object-creation operations. When a new `inode` is created, the security module allocates space for the `inode` security field with the `alloc_security()` inode hook. Either post-lookup or post-creation, the newly created objects are labelled. The label may be cleared by the `delete()` inode hook when an inode's link count reaches zero. And finally, when an `inode` is destroyed, the `free_security()` inode hook is called to release the space allocated for the security field.

[4]In some cases, `super_blocks` are recycled.

In many cases, the LSM inode hooks are identical to the `inode_operations`. For all `inode_operations` that can create new filesystem objects a "post" inode hook is defined for coherent security labelling. For example, when a task creates a new symlink, the `symlink()` inode hook is called to check permission to create the symlink. Then if the symlink creation is successful, the `post_symlink()` hook is called to set the security label on the newly created symlink.

Whenever possible, LSM leverages the existing Linux kernel security infrastructure. The kernel's standard UNIX DAC checks compare the uids, guids, and mode bits when checking for permission to access filesystem objects. The VFS layer already has a `permission()` function which is a wrapper for the `permission() inode_operation`. LSM uses this pre-existing infrastructure and adds its `permission()` inode hook to the VFS wrapper.

3.4.3 File Hooks

The kernel's `file` structure represents an open filesystem object. It contains the `file_operations` structure, which describes the operations that can be done to a `file`. For example, a `file` can be read from and written to, seeked through, mapped into memory, and so on. Similar to the inode hooks, LSM provides file hooks to mediate access to `files`, many of which mirror the `file_operations`. A security field has been added to the `file` structure for labelling.

When a file is opened, a new `file` object is created. At this time, the `alloc_security()` file hook is called to allocate a security field and label the `file`. This label persists until the `file` is closed, when the `free_security()` file hook is called to free the security field.

The `permission()` file hook can be used to revalidate read and write permissions at each `file` read or write. This is not effective against reading and writing of memory mapped files, and the changes required to support this page level revalidation are considered too invasive. Actually mapping a file is, however, protected with the `mmap()` file hook. And changing the protection bits on mapped file regions must pass the `mprotect()` file hook.

When using `file` locks to synchronize multiple readers or writers, a task must pass the `lock()` file hook permission check before performing any locking operation on a `file`.

If the `O_ASYNC` flag is set on a `file`, asynchronous I/O ready signals are delivered to the `file` owner when the `file` is ready for input or output. The ability to specify the task that will receive the I/O ready signals is protected by the `set_fowner()` file hook. Also, the actual signal delivery is mediated by the `send_sigiotask()` file hook.

Miscellaneous `file` operations that come through the `ioctl(2)` and `fcntl(2)` interfaces are protected by the `ioctl()` and `fcntl()` file hooks respectively. Another miscellaneous action protected by the file hooks is the ability to receive an open file descriptor through a socket control message. This action is protected by the `receive()` file hook.

3.5 IPC Hooks

The Linux kernel provides the standard SysV IPC mechanisms: shared memory, semaphores, and message queues. LSM defines a set of IPC hooks which mediate access to the kernel's IPC objects. Given the design of the kernel's IPC data structures, LSM defines one common set of IPC hooks, `ipc_security_ops`, as well as sets of object specific IPC hooks: `shm_security_ops`, `sem_security_ops`, `msg_queue_security_ops`, and `msg_msg_security_ops`.

3.5.1 Common IPC Hooks

The kernel's IPC object data structures share a common credential structure, `kern_ipc_perm`. This structure is used by the kernel's `ipcperms()` function when checking IPC permissions. LSM adds a security field to this structure and an `ipc_security_ops` hook, `permission()`, to `ipcperms()` to give the security module access to these existing mediation points. LSM also defines an `ipc_security_ops` hook, `getinfo()`, to mediate info requests for any of the IPC objects.

3.5.2 Object Specific IPC Hooks

The LSM IPC object specific hooks define the `alloc_security()` and `free_security()` functions to manage the security field in each object's `kern_ipc_perm` data structure. An IPC object is created with an initial "get" request, which triggers the object specific `alloc_security`. If the "get" request finds an already existing object, the `associate()` hook is called to check permissions before returning the object.

IPC object control commands, `shmctl(2)`, `semctl(2)`, and `msgctl(2)` are mediated by object specific "ctl" hooks. For example, when a `SHM_LOCK` request is issued, the `shm_security_ops` `shmctl()` hook is checked for permission prior to completing the request.

Any attempt to change a semaphore count is protected by the `sem_security_ops` `semop()` hook. Attaching to a shared memory segment is protected by the `shm_security_ops` `shmat()` hook. Sending and receiving messages on a message queue are protected by the `msg_queue_security_ops` `msgsnd()` and `msgrcv()` hooks. The individual messages are considered as well as the queue when verifying permission. When a new message is created, the `msg_msg_security_ops` `alloc_security()` hook allocates the security field stored in the actual message data structure. Upon receipt, the `msgrcv()` hook can verify the security field on both the queue and the message.

3.6 Module Hooks

The LSM interface would surely be incomplete if it didn't mediate loading and unloading kernel modules. The LSM module loading hooks, `module_security_ops`, add permission checks protecting the creation and initialization of loadable kernel modules as well as module removal.

3.7 Network Hooks

The Linux kernel features an extensive suite of network protocols and supporting components. As networking is an important aspect of Linux, LSM extends the concept of a generalized security framework to this area of the kernel.

A key implementation challenge was to determine the initial requirements for the network hooks. The existing SELinux implementation was utilized as a model, as SELinux is itself a highly generalized security infrastructure which was to be ported to LSM. Other Linux security projects were reviewed, although none relevant to the version 2.5 kernel series were found with networking requirements in excess of SELinux. Potential requirements for IPSec and traditional labeled networking systems were also taken into account.

As the Linux network stack utilizes the Berkeley sockets model [21], LSM is able to provide coarse coverage for all socket-based protocols via the use of hooks within the socket layer.

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.

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. Accesses to these objects can be interposed at higher levels (e.g., via system calls such as `ioctl(2)`), although granularity may be reduced by TOCTTOU issues. The existing kernel code does however impose a `CAP_NET_ADMIN` capability requirement for tasks which attempt to write to important network support components.

The details of the network hooks are described in Sections 3.7.1 through 3.7.6.

3.7.1 Sockets and Application Layer

Application layer access to networking is mediated via a series of socket-related hooks, `socket_security_ops`. When an application attempts to create a socket with the `socket(2)` system call, the `create()` hook allows for mediation prior to the actual creation of the socket. Following successful creation, the `post_create()` hook may be used to update the security state of the inode associated with the socket.

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. However, it is possible for sockets to temporarily exist in a state where they have no `socket` or `inode` structure. Hence, the networking hook functions must take care in extracting the security information for sockets.

Mediation hooks are also provided for all of the socket system calls:

```
bind(2)
connect(2)
listen(2)
accept(2)
sendmsg(2)
recvmsg(2)
getsockname(2)
getpeername(2)
getsockopt(2)
setsockopt(2)
shutdown(2)
```

Protocol-specific information is available via the `socket` structure passed as a parameter to all of these hooks (except for `create()`, as the socket does not yet exist at this hook). This facilitates mediation based on transport layer attributes such as TCP connection state, and seems to obviate the need for explicit transport layer hooks.

The `sock_rcv_skb()` hook is called when an incoming packet is first associated with a socket. This allows for mediation based upon the security state of receiving application and security state propagated from lower layers of the network stack via the `sk_buff` security field (see section 3.7.2).

Additional socket hooks are provided for Unix domain communication within the abstract namespace, as binding and connecting to Unix domain sockets in the abstract namespace is not mediated by filesystem permissions. The `unix_stream_connect()` hook allows mediation of stream connections, while datagram based communications may be mediated on a per-message basis via the `unix_may_send()` hook.

3.7.2 Packets

Network data traverses the network stack in packets encapsulated by a structure called an `sk_buff` (socket buffer). The `sk_buff` structure provides storage for packet data and related state information, and is considered to be owned by the current layer of the network stack.

LSM adds an opaque 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 the security field. For LSM, the critical lifecycle events for an `sk_buff` are:

- Allocation
- Copying
- Cloning
- Setting ownership to sending socket
- Datagram reception
- Destruction

Hooks are provided for each of these events, although they are only intended to be used for maintaining the security field data. Encoding, decoding and interpretation of the security field data is performed by layer-specific hooks such as the socket and network layer hooks.

Generally, the `sk_buff` hooks and security field only need to be used when the security state of a packet must be managed between layers of the network stack. Examples of such cases include labeled networking via IP options and management of nested IPSec Security Associations [15].

3.7.3 Transport Layer (IPv4)

Explicit hooks are not required for the transport layer, as sufficient protocol state information for LSM is available at the socket and network layer hooks (discussed in section 3.7.1).

3.7.4 Network Layer (IPv4)

Hooks are provided at the network layer for IPv4 to facilitate:

- Integrated packet filtering
- IP options decoding for labeled networking
- Management of fragmented datagrams
- Network layer encapsulation (e.g., secure IP tunnels)

Existing Netfilter [23] hooks are used to provide access to IP datagrams in pre-routing, local input, forwarding, local output and post-routing phases. Through these hooks, LSM intercepts packets before and after the standard iptables-based access control and translation mechanisms. Note that the Netfilter hooks used by LSM do not increase the code footprint imposed by LSM on the standard kernel.

3.7.5 Network Devices

Within the Linux network stack, hardware and software network devices are encapsulated by a `net_device` structure. LSM adds an security field to this structure so that security state information can be maintained on a per-device basis.

The security field for the `net_device` structure may be allocated during first-use initialization. A security field management hook is called when the device is being destroyed, allowing any allocated resources associated with the associated security field to be freed.

3.7.6 Netlink

Netlink sockets are a Linux-specific mechanism for kernel-userspace communication. They are similar to BSD route sockets, although more generalized.

As Netlink communications are connectionless and asynchronously processed, security state associated with an application layer origin needs to be stored with Netlink packets, then checked during delivery to the destination kernel module. The `netlink_send()` hook is used to store the application layer security state. The `netlink_recv()` hook is used to retrieve the stored security state as the packet is received by the destination kernel module and mediate final delivery.

3.8 Other System Hooks

LSM defines a miscellaneous set of hooks to protect the remaining security sensitive actions that are not covered by the hooks discussed above. These hooks typically mediate system-level actions such as setting the system's host name or domain name, rebooting the system, and accessing I/O ports. The existing capability checks already protect these actions; however, the LSM hooks provide more finely grained access control.

The LSM interface leverages the pre-existing POSIX.1e capabilities infrastructure in the Linux kernel. The capability checks can often override standard DAC checks (akin to root). The checks are limited to a 32 bit vector describing the required capability, e.g., `CAP_DAC_OVERRIDE`, and thus give the module limited context when making access control decisions. The system-level `capable()` hook is placed in the existing `capable()` function which gives LSM easy compatibility with POSIX.1e capabilities as well as a moderate ability to override DAC checks.

The LSM framework adds a security system call, which is a thin wrapper around the `sys_security()` hook in the LSM interface. This system call is a simple multiplexor which allows a module to define a set of policy specific system calls. The LSM security system call interface is modeled after the standard Linux socket system call multiplexor, `sys_socketcall(2)`.

4 Testing and Functionality

The true impact of LSM will be felt if and when LSM is accepted as a standard part of the Linux kernel, and end-users can adopt security modules as readily as they adopt other applications for Linux. To be accepted into Linux, LSM must be highly cost-effective. Section 4.1 summarizes the performance cost of the LSM infrastructure. Section 4.2 presents the security impact of LSM, in the form of modules that have already been implemented or ported to LSM.

4.1 Performance Impact

The performance cost of the LSM framework is critical to its acceptance; in fact, performance cost was a major part of the debate at the Linux 2.5 developer's summit that spawned LSM. To rigorously document the performance costs of LSM, we performed both microbenchmarks and macrobenchmarks that compared a stock Linux kernel to one modified with the LSM patch, but with no modules loaded.[5]

For microbenchmarks, we used the LMBench [22] tool. 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.

LMBench outputs prodigious results. 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. The common case was much better, often 0% overhead, ranging up to 2% overhead.

For macrobenchmarking, we used the common approach of building the Linux kernel from source. The results here were even better: no measurable performance impact.[6] More detailed performance data can be found in [31].

4.2 Security Impact

Another key factor in the acceptance of the LSM framework is that it provide some real security value. This can be viewed in two ways. First, LSM must not create new security holes and needs to be thorough and consistent in its coverage. Second, the LSM framework must be general enough to support a variety of access control models.

Proving the correctness of the LSM framework has not been handled by the LSM project directly. How-

[5] The performance costs of each module are the responsibility of the module's authors.

[6] In fact, the LSM case was actually faster, but we regard that as an experimental anomaly, and do not claim that LSM is a performance optimization :-)

ever, a project from IBM [9] has developed tools to do both static and dynamic analysis of the LSM framework. These tools have, in fact, helped improve the LSM interface, and can help with ongoing maintenance.

The real value of LSM is delivering effective security modules. Porting access control models to the LSM framework proves that it is functional as a general purpose access control framework. As the name suggests, LSM does not impact system security without security modules. Presently, LSM supports the following security modules:

- **SELinux** A Linux implementation of the Flask [28] 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 [19] 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 [3, 4] developed for Linux [14]. 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 [8] 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 [29] 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.
- **LIDS (Linux Intrusion Detection System)** started out as an intrusion detection system, and then migrated towards intrusion prevention in the form of an access control system similar to SubDomain [7] that manages access by describing what files a given *program* may access.

5 Conclusions

Linux is a shared playroom, and thus needs to make most players reasonably happy. LSM thus needs to meet two criteria: be relatively painless for people who don't want it, and be useful and effective for people who do want it.

We feel that LSM meets these criteria. The patch is relatively small, and the performance data in Section 4 shows that the LSM patch imposes nearly zero overhead. The broad suite of security products from around the world that have been implemented for LSM shows that the LSM API is useful and effective for developing Linux security enhancements.

6 Acknowledgements

Thanks to the LSM mailing list for engaging in the sometimes tedious and heated discussions that helped shape LSM. Special thanks to the SELinux project that helped kickstart LSM with the original presentation at the 2001 Kernel Summit.

7 Availability

LSM is available as a kernel patch for both the 2.4 and 2.5 Linux kernels. The patches are available from `http://lsm.immunix.org`.

References

[1] The Holy Bible: Genesis 11:1-8.

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

[3] 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.

[4] 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.

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

[6] 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`.

[7] 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.

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

[9] Antony Edwards and Xiaolan Zhang. Using CQUAL for Static Analysis of Authorization Hook Placement. In *USENIX Security Symposium*, San Francisco, CA, August 2002.

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

[11] 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.

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

[13] 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.

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

[15] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] S. J. Leffler W. N. Joy M. K. McKusick, M. J. Karels and R. S. Faber. *Berkeley Software Architecture Manual, 4.4BSD Edition.* University of California, Berkeley, Berkeley, CA, 1994.

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

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

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

[25] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, 1999. `http://www.oreilly.com/catalog/cb/`.

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

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

[28] 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.

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

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

[31] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, San Francisco, CA, August 2002.

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