Implementing SELinux as a Linux Security Module

Stephen Smalley
NSA

sds@epoch.ncsc.mil

Chris Vance
NAI Labs

cvance@nai.com

Wayne Salamon
NAI Labs

wsalamon@nai.com

This work supported by NSA contract MDA904-01-C-0926 (SELinux)
Initial: December 2001, Last revised: Feb 2006
NAI Labs Report #01-043

Table of Contents

1. Introduction

In March 2001, the National Security Agency (NSA) gave a presentation about Security-Enhanced Linux (SELinux) at the 2.5 Linux Kernel Summit. SELinux is an implementation of flexible and fine-grained nondiscretionary access controls in the Linux kernel, originally implemented as its own particular kernel patch. The design and implementation of the original SELinux prototype is described in [LoscoccoFreenix2001] and [LoscoccoNSATR2001], both of which can be found at the NSA SELinux web site (http://www.nsa.gov/selinux).

In response to the NSA presentation, Linus Torvalds made a set of remarks that described a security framework he would be willing to consider for inclusion in the mainstream Linux kernel. He described a general framework that would provide a set of security hooks to control operations on kernel objects and a set of opaque security fields in kernel data structures for maintaining security attributes. This framework could then be used by loadable kernel modules to implement any desired model of security.

The Linux Security Modules (LSM) project was started by Immunix to develop such a framework. LSM was a joint development effort by several security projects, including Immunix, SELinux, SGI and Janus, and several individuals, including Greg Kroah-Hartman and James Morris, to develop a Linux kernel patch that implements this framework. The LSM framework is included as part of the Linux 2.6 series.

The SELinux implementation was adapted to use the LSM framework rather than its own particular kernel patch. This technical report documents the LSM-based SELinux security module. The report begins by providing an overview of LSM and a review of the SELinux basic concepts. It then provides an overview of how the LSM-based SELinux security module differs from the original SELinux kernel patch. Several aspects of the SELinux security module are then described, including its internal architecture, its initialization code, its support for stacking with other security modules, and its approach for implementing the SELinux API. The remainder of the report is then spent documenting the SELinux hook function implementations, organized into sections for each grouping of LSM hooks. Typically, these hooks are grouped based on the relevant kernel object or kernel subsystem.

* *Note: This report predates modern enhancements to the SELinux kernel code, such as the introduction of labeled networking support (labeled IPSEC and NetLabel/CIPSO), the introduction of APIs for getting peer and datagram security contexts for INET*

and Unix socket IPC, and significant changes to the SELinux network access controls. Thus, while much of the discussion herein is still applicable, much has changed in modern SELinux kernels.

2. Acknowledgements

We thank James Morris for his contributions to the SELinux security module and for his independent development of CIPSO/FIPS188 packet labeling for SELinux. We thank the other contributors to the LSM kernel patch for their work, particularly Chris Wright, Greg Kroah-Hartman, James Morris, Serge Hallyn, and Lachlan McIlroy. We also thank the users of SELinux for their feedback on the LSM-based SELinux releases.

3. LSM Overview

This section provides an overview of the Linux Security Modules (LSM) framework. This section contains an edited excerpt from the `Documentation/DocBook/lsm.tmpl` file in the kernel tree, updated to reflect recent changes made for the Linux 2.6 integration.

LSM provides a general kernel framework to support security modules. In particular, the LSM framework is primarily focused on supporting access control modules. By itself, the framework does not provide any additional security; it merely provides the infrastructure to support security modules. The LSM framework also moves most of the capabilities logic into an optional capabilities security module, with the system defaulting to a dummy security module that implements the traditional superuser logic.

The LSM framework adds security fields to kernel data structures and inserts calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. It also adds functions for registering and unregistering security modules. Extended attribute handlers for a new security namespace were added to filesystems to support new file security attributes, and a `/proc/pid/attr` subdirectory was introduced to provide userspace access to new process security attributes.

The LSM security fields are simply void* pointers. For process and program execution security information, security fields were added to struct task_struct and struct linux_binprm. For filesystem security information, a security field was added to struct super_block. For pipe, file, and socket security information, security fields were added to struct inode and struct file. Unix domain sockets may also use a security field added to the struct sock. For System V IPC security information, security fields were added to struct kern_ipc_perm and struct msg_msg.

Each LSM hook is a function pointer in a global table, security_ops. This table is a security_operations structure as defined by `include/linux/security.h`. Detailed documentation for each hook is included in this header file. The hooks are grouped into logical sets based on the kernel object (e.g. task, inode, file, sock, etc) as well as some miscellaneous hooks for system operations. A static inline function is defined for each hook, so that most of the hook calls can easily be compiled away if desired, in which case only the default capabilities logic is included.

The global security_ops table is initialized to a set of hook functions provided by a dummy security module that provides traditional superuser logic. A `register_security` function (in `security/security.c`) is provided to allow a security module to set security_ops to refer to its own hook functions, and an `unregister_security` function is provided to revert security_ops to the dummy module hooks. This mechanism is used to set the primary security module, which is responsible for making the final decision for each hook.

LSM also provides a simple mechanism for stacking additional security modules with the primary security module. It defines `register_security` and `unregister_security` hooks in the security_operations structure and provides `mod_reg_security` and `mod_unreg_security` functions that invoke these hooks after performing some sanity checking. A security module can call these functions in order to stack with other modules. However, the actual details of how this stacking is handled are deferred to the module, which can implement these hooks in any way it wishes (including always returning an error if it does not wish to support stacking). In this manner, LSM defers the problem of composition to the module.

Although the LSM hooks are organized based on kernel object, all of the hooks can be viewed as falling into two major categories: hooks that are used to manage the security fields and hooks that are used to perform access control. Examples of the first category of hooks include the `alloc_security` and `free_security` hooks defined for each kernel data structure that has a security field. These hooks are used to allocate and free security structures for kernel objects. The first category of hooks also includes hooks that set information in the security field after allocation, such as the `d_instantiate` hook. This hook is used to set security information for inodes, e.g. by calling getxattr to obtain an attribute value, when all the necessary object information is available. An example of the second category of hooks is the `inode_permission` hook. This hook checks permissions when accessing an inode.

Although LSM originally included a new security system call, this call was subsequently removed. Most of its functionality can now be implemented using the extended attribute support and `/proc/pid/attr` interface, as mentioned above.

4. SELinux Basic Concepts

This section provides an overview of the SELinux basic concepts. More background information about SELinux can be found in [LoscoccoFreenix2001].

SELinux is based on the Flask security architecture for flexible nondiscretionary access controls. This architecture was previously implemented in the Fluke research operating system, as described in [SpencerUsenixSec1999]. The Flask security architecture provides a clean separation between the policy enforcement code and the policy decision-making code. The policy decision-making code is encapsulated in a separate component of the operating system called the security server. The Flask security architecture includes an access vector cache (AVC) component that provides caching of access decision computations obtained from the security server to minimize the performance overhead of the SELinux access controls. The policy enforcement code is integrated into the subsystems (e.g. the process management code, the filesystem code, the socket and networking code, and the IPC code) of the operating system. The policy enforcement code obtains security policy decisions from the security server and AVC, and applies those decisions to assign security labels to processes and objects and to control operations based on those security labels.

Since different security policies require different kinds of security attributes, the Flask security architecture provides two policy-independent data types for security labels: the security context and the security identifier (SID). A security context is a string representation of a security label, while a SID is an internal handle that is mapped by the security server to a security context. Kernel SIDs are not exported to userspace; the kernel only returns security contexts to userspace. However, userspace policy enforcers may have their own SID mappings maintained by the userspace AVC that is included in libselinux. Both SIDs and security contexts are handled opaquely by the policy enforcement code and can only be interpreted by the security server. The policy enforcement code binds SIDs to active processes and objects, consulting the security server when a SID needs to be computed for a new subject or object. The policy enforcement code in the filesystem code also stores file security contexts in each filesystem using extended attributes.

The policy enforcement code consults the AVC to check permissions for operations, passing a pair of SIDs and a security class; the AVC obtains access decisions from the security server as needed. The pair of SIDs are referred to as a source SID and a target SID. Typically, the source SID is the SID of a process and the target SID is the SID of another process or an object, but it is also possible for permissions to be defined between two objects to control relationships among objects. The security class identifies the kind of object. Each security class has an associated set of permissions that are used to control access to that object. These permission sets are represented by a bitmap called an access vector.

5. Changes from the Original SELinux Kernel Patch

This section summarizes the changes between the original SELinux kernel patch and the LSM-based SELinux security module. At a high level, the LSM-based SELinux security module provides equivalent security functionality to the original SELinux kernel patch. However, there have been some changes to the specific controls, partly driven by design constraints imposed by LSM and partly based on further review of the original SELinux controls. There have also been significant changes in the underlying implementation, likewise partly driven by differences in LSM and partly based on a review of the original SELinux implementation. The following subsections summarize the changes, grouped by category.

5.1. General Changes

This subsection describes general changes between the original SELinux kernel patch and the LSM-based SELinux security module. These changes include adding a new level of indirection, dynamically allocating security fields, stacking with the capabilities module, redesigning the SELinux API, and leveraging the existing Linux functions for checking permissions.

5.1.1. Adding a New Level of Indirection

The original SELinux kernel patch provided clean separation between the policy enforcement code and the policy decision-making code by using the Flask security architecture and interfaces. The policy enforcement code was directly inserted into the kernel code at appropriate points, and the policy decision-making code was encapsulated in the security server, with a well-defined interface between the two components. Similarly, policy-independent data types for security information were directly inserted into kernel data structures, and only the security server could interpret these data types. This level of separation permitted many different kinds of nondiscretionary access control policies to be implemented in the security server without any changes to the policy enforcement code.

The LSM kernel patch inserts calls to hook functions on kernel objects into the kernel code at appropriate points, and it inserts void* security fields into the kernel data structures for kernel objects. In the LSM-based SELinux security module, the policy enforcement code is implemented in the hook functions, and the policy-independent data types are stored using the security fields in the kernel data structures. Internally, the SELinux code continues to use the Flask architecture and interfaces, and the security server remains as a separate component of the module. Hence, LSM introduces an additional level of indirection for the SELinux code and data. The internal architecture of the SELinux security module is discussed further in Section 6.

5.1.2. Dynamically Allocating Security Fields

In the original SELinux kernel patch, fields for security data were inserted directly into the appropriate kernel objects and were allocated and freed with the kernel object. Since LSM inserts only a single void* security field into each kernel object, the LSM-based SELinux security module must manage a dynamically allocated security structure for each kernel object unless it only needs to store a single word of security data. The SELinux security module uses a dynamically-allocated security structure for the security fields of the kernel data structures.

5.1.3. Stacking with the Capabilities Module

The original SELinux kernel patch added the SELinux nondiscretionary access controls as additional restrictions to the existing Linux access control logic. This left the existing Linux logic intact and unchanged, including the discretionary access control logic and the capabilities logic. LSM moves most of the capabilities logic into an optional capabilities security module and provides a dummy security module that implements traditional superuser logic. Hence, the LSM-based SELinux security module provides support for stacking with either the capabilities module or the dummy module. Since some existing applications (e.g. named, sendmail) expect capabilities to be present in Linux, it is recommended that the SELinux module always be stacked with the capabilities module. The stacking support is discussed further in Section 8.

5.1.4. Redesigning the SELinux API

In the original SELinux kernel patch, extended system calls such as `execve_secure` and `stat_secure` were implemented by extending the internal kernel functions to optionally pass and process SID parameters. Initially, in the LSM-based SELinux security module, these extended system calls were implemented using the security system call and by passing SID parameters to and from the hook functions via fields in the current task's security structure. However, when the security system call was removed from LSM, the SELinux API was completely redesigned in order to gain acceptance into the mainline kernel. This is discussed further in Section 9.

5.1.5. Leveraging Linux Permission Functions

The original SELinux kernel patch directly inserted its own permission checks throughout the kernel code rather than trying to leverage existing Linux permission functions such as `permission` and `ipcperms` due to the coarse-grained permissions supported by these functions and the need to perform permission checks in many locations where no Linux check already existed. The one notable exception to this practice in the original SELinux kernel patch was the insertion of a SELinux permission check into the existing `capable` kernel function so that SELinux could perform a parallel check for the large number of existing calls to `capable`.

In contrast, LSM inserts hook calls into all of the existing Linux permission functions in order to leverage these functions. In some cases, LSM also inserts additional hook calls in specific operations to provide finer-grained control, but in other cases, it merely relies on a hook in one of the existing Linux permission functions to control an operation. The LSM-based SELinux security module uses the hooks in the existing Linux permission functions to perform a parallel check for each Linux permission check. These parallel checks for the Linux permission checks ensure that every Linux access control is also controlled by SELinux. They also reduce the risk that future changes to Linux will introduce operations that are completely uncontrolled by SELinux.

Using these hooks required defining some additional coarse-grained permissions for SELinux. These permissions are discussed further in Section 5.3.3 and in Section 5.5.2. Whenever possible, the LSM-based SELinux security module leverages these hooks to provide control. When SELinux requires finer-grained control, the module implements these finer-grained SELinux controls using the additional LSM hooks.

5.2. Program Execution Changes

This subsection describes general changes between the original SELinux kernel patch and the LSM-based SELinux security module related to program execution. These changes include replacing the process `execute` permission with a new file `execute_no_trans` permission, and changing the controls over the inheritance of state across a context-changing execve. Each of these changes is described below.

5.2.1. File `execute_no_trans` Permission

In the original SELinux kernel patch, the file `execute` permission controlled the ability to initiate the execution of a program, while the process `execute` permission controlled the ability to execute code from an executable image. The distinction was necessary because the SID of a task can be changed by program execution, so the SID of the initiator may differ from the SID of the transformed process. However, the process `execute` permission was redundant with the process `entrypoint` permission when the SID of the task was changing, so it only served a useful purpose when the task SID was left unchanged. Furthermore, since this permission was between a task SID and a program file SID, it properly belonged in the file class, not the process class.

Hence, the process `execute` permission was replaced by a new file `execute_no_trans` permission in the LSM-based SELinux security module. Unlike the original process `execute` permission, the file `execute_no_trans` permission is only checked when the SID of the task would remain unchanged. The process `entrypoint` permission was also moved into the file class for consistency. The file `execute` and process `transition` permissions were left unchanged. These checks are described further in Section 12.1.3.

5.2.2. Inheritance of State

Several changes were made to the controls over the inheritance of state across a context-changing execve. These changes included changes to the file descriptor inheritance controls, changes to the controls over process tracing and state sharing, and the addition of new controls.

The file descriptor inheritance permission checks during program execution were revised for the LSM-based SELinux security module. This is discussed in Section 5.3.4.

In the original SELinux kernel patch, checks for process tracing and sharing process state when the SID was changed were inserted into the `compute_creds` kernel function with the existing Linux tests for these conditions for setuid programs. However, this function can not return an error, so SELinux merely left the task SID unchanged if these checks failed, just as Linux leaves the uid unchanged if its tests fail. Additionally, the original SELinux kernel patch used a hardcoded test for process 1 to permit the kernel to transition to a new SID for `init` even though it was sharing state. In the LSM-based SELinux security module, the ptrace and share checks were changed to also send a SIGKILL to the task to terminate it upon a permission failure, and a new process `share` permission was added to provide configurable control over process state sharing across SID transitions. This is described further in Section 12.1.4.

New permission checks were implemented in the LSM-based SELinux to control inheritance of signal-related state and resource limits. These checks are also described in Section 12.1.4. Furthermore, a `AT_SECURE` flag was added to the ELF auxiliary table so that the SELinux module could inform glibc when to enable its own secure mode in order to sanitize the environment and other state on a context-changing exec. This behavior is also controlled based on a permission check between the relevant contexts, and is described in Section 12.1.6.

5.3. Filesystem Changes

This subsection describes changes between the original SELinux kernel patch and the LSM-based SELinux security module related to the filesystem. These changes include using extended attributes rather than the persistent label mapping for file security contexts on persistent filesystems, reimplementing file labeling support for pseudo filesystem types, leveraging the hook in the existing `permission` function, revising the file descriptor permission checks, and eliminating the pipe security class. Each change is described below.

5.3.1. Labeling of Persistent Files

In the original SELinux kernel patch, a persistent label mapping was maintained in each filesystem that stored a mapping from integer persistent security identifiers (PSIDs) to security contexts, and a PSID was stored in a spare field of the on-disk ext2 inode. Since LSM provides all of its file-related hooks in the VFS layer and does not provide any filesystem-specific hooks, the SELinux persistent label mapping was initially changed to maintain the inode-to-PSID mapping in a regular file rather than using a spare field in the ext2 on-disk inode. This change allowed SELinux to support other file system types more easily, but had disadvantages in terms of performance and consistency. Since support for extended attributes was integrated into the Linux 2.6 kernel, extended attribute handlers were created for a new security namespace, and SELinux was modified to store file security contexts as extended attributes. This eliminated the need for the persistent label mapping.

5.3.2. Pseudo Filesystem Labeling

In the original SELinux kernel patch, code was directly inserted into the procfs and devpts pseudo filesystem implementations to provide appropriate file labeling behaviors. Since LSM did not provide filesystem-specific hooks, the LSM-based SELinux security module had to reimplement this functionality using the hooks in the VFS layer. Subsequently, as part of the integration of SELinux into Linux 2.6, a LSM hook was introduced into the proc filesystem to better support labeling of /proc/pid inodes, and a fake xattr handler was added to the devpts pseudo filesystem implementation to export pty labels to userspace. However, labeling of other proc inodes and the initial labeling of devpts inodes is still handled by the hooks called by the VFS layer. The LSM-based SELinux also expanded and generalized support for pseudo filesystem labeling. The handling for these pseudo filesystem types is described in Section 14.1.3.

5.3.3. Leveraging `permission`

As discussed in Section 5.1.5, LSM inserts a hook into the existing Linux functions for permission checking, including the `permission` function for checking access to objects represented by inodes. The LSM-based SELinux security module leverages this hook to perform a parallel check for each existing Linux inode permission check. The use of this hook posed a problem for preserving the SELinux distinction between opening a file with append access vs. opening a file with write access, requiring an additional change to the Linux kernel.

The use of this hook also posed a problem for the SELinux directory permissions, which partition traditional write access into separate permissions for adding entries (`add_name`), removing entries (`remove_name`), and reparenting the directory (`reparent`). Since these distinctions are not possible in the `selinux_inode_permission` hook called by the `permission` kernel function, a directory `write` permission was added to SELinux. This permission is checked by this hook when write access is requested, and the finer-grained directory permissions are checked by the additional hooks that are called when a directory operation is performed.

Hence, directory modifications require both a `write` permission and the appropriate finer-grained permission to the directory. Whenever one of the finer-grained permissions is granted in the policy, the `write` permission should also be granted in the policy. The `write` permission check on directories could be omitted, but it is present to ensure that all directory write accesses are controlled by SELinux.

5.3.4. File Descriptor Permissions

In the original SELinux kernel patch, distinct file descriptor permissions were defined for getting the file offset or flags (`getattr`), setting the file offset or flags (`setattr`), inheriting the descriptor across an `execve` (`inherit`), and receiving the descriptor via socket IPC (`receive`). These permissions were reduced to a single `use` permission in the LSM-based SELinux security module that is checked whenever the descriptor is inherited, received, or used.

Additionally, in the original SELinux kernel patch, only the `inherit` or `receive` permissions were checked when a descriptor was inherited or received. The other descriptor permissions and the appropriate file permissions were only checked when an attempt was made to use the descriptor. In the LSM-based SELinux security module, the `use` permission and the appropriate file permissions are checked whenever the descriptor is inherited, received, or used.

These changes to the SELinux file descriptor permission checks bring SELinux into conformity with the base Linux control model, where possession of a descriptor implies the right to use it in accordance with its mode and flags. This reduces the risk of misuse of a descriptor by a process, and also reduces the risk that future changes to Linux will open vulnerabilities in the SELinux control model. With these changes, the SELinux permission checks on calls such as `read` and `write` are only necessary to support revocation of access for relabeled files or policy changes.

5.3.5. Pipe Security Class

In the original SELinux kernel patch, a separate security class was defined for pipes, although this security class merely inherited the common file permissions. In the LSM-based SELinux security module, this class was eliminated, and the `fifo_file` security class is used for both pipes and for named FIFOs. This has no impact on the ability to control pipe operations distinctly, since pipes are still labeled with the SID of the creating task while named FIFOs are labeled in the same manner as other files.

5.4. Socket IPC and Networking Changes

This subsection describes changes between the original SELinux kernel patch and the LSM-based SELinux security module related to socket IPC and networking. These changes include redesigning the SELinux network access controls, storing socket security information in the associated inode security field, reimplementing the SELinux access controls using minimally invasive hooks, changing the file descriptor transfer controls, omitting some of the low-level `ioctl` controls, and implementing the extended socket calls.

5.4.1. Redesigning Network Access Controls

As part of integrating SELinux into Linux 2.6, the network access controls were redesigned based on past experience and on what could be readily supported by the Linux 2.6 kernel, since most of the LSM networking hooks were rejected. This is discussed further in Section 17.2.3 and Section 18.

5.4.2. Storing Socket Security Data

The original SELinux kernel patch added security fields to the network layer sock structure for socket security data, and also mirrored the SID and security class of the socket in the inode structure associated with the socket. LSM also provides a security field within the sock structure, but SELinux can only use this field to store peer security data for Unix stream connections during connection setup. Otherwise, the LSM-based SELinux security module stores all socket security data in the security field of the associated inode once the user socket is established. This is discussed further in Section 17.1 and Section 18.

5.4.3. Minimally Invasive Hooks

Since the original SELinux kernel patch added security fields to the lower-level struct sock structure, most of the SELinux changes were inserted directly into the specific protocol family implementations (e.g. the AF_INET and AF_UNIX code). The original SELinux kernel patch was fairly invasive in inserting SELinux processing throughout the protocol family implementations, and did not try to leverage the existing Linux packet filtering support.

LSM provides a set of hooks in the abstract socket layer for controlling socket operations at a high level, and leverages the Linux NetFilter support for hooking network operations. The LSM-based SELinux security module implements as many of the SELinux socket and network controls as possible using these socket layer hooks and NetFilter-based hooks. Hence, NetFilter support should be enabled in the kernel configuration when using SELinux.

For the SELinux Unix domain IPC controls, the LSM-based SELinux security module leverages the hooks in the existing Linux permission functions but also required two additional hooks in the Unix domain protocol implementation due to the abstract namespace. The SELinux socket access controls are described in Section 17.2 and the SELinux network layer access controls are described in Section 18.

5.4.4. File Descriptor Transfer

The file descriptor transfer permission checks during socket IPC were revised for the LSM-based SELinux security module. This is discussed in Section 5.3.4.

5.4.5. Omitting Low-Level `ioctl` Controls

In the original SELinux kernel patch, a small set of controls were implemented in low-level `ioctl` routines to support fine-grained control over configuring network devices, accessing the kernel routing table, and accessing the kernel ARP and RARP tables. During the development of LSM, the feasibility of providing hooks to support these controls was explored, but it was determined that providing hooks in every location necessary to control configuring network devices would be too invasive, and the other controls offered little benefit over the existing `capable` calls. Hence, the LSM-based SELinux security module does not implement these controls, and control over these operations is handled based on the `capable` calls.

5.4.6. Extended Socket Calls

In the original SELinux kernel patch, a set of extended socket calls were implemented. These calls were reimplemented initially for the LSM-based SELinux, and an experimental labeled networking implementation was also contributed. However, as part of the SELinux API redesign and the rejection of the LSM networking hooks, the extended socket calls and labeled networking do not exist in Linux 2.6. There is one exception: a getpeercon API has been implemented to support obtaining peer security contexts for Unix stream connections, and is available in Linux 2.6.

* *Note: The preceding statements are historical and no longer apply to modern SELinux systems, which do support labeled networking and APIs for getting peer and datagram contexts on both INET and Unix sockets.*

5.5. System V IPC Changes

This subsection describes changes between the original SELinux kernel patch and the LSM-based SELinux security module related to System V IPC. Since the System V IPC security enhancements were never ported from the 2.2 series to the 2.4 series prior to the transition to using LSM, the LSM-based SELinux security module had to adapt the implementation of the SELinux security enhancements to the

2.4 series. In addition to this adaptation, the changes include an easier solution for storing the IPC security data and leveraging the hook in the existing `ipcperms` function.

5.5.1. Storing IPC Security Data

In the original SELinux kernel patch for the 2.2 series, it was difficult to add security data to the semaphore and message queue structures because the kernel exported the same data structure that it used internally to applications. Hence, the original SELinux kernel patch wrapped these data structures with private kernel data structures that contained both the original structure and the additional security data. This required extensive changes to the IPC code to dereference fields in the original structure. In the 2.4 series, the IPC code was rewritten to use private kernel data structures for all of the IPC objects, and each of these structures included a struct kern_ipc_perm structure with common information. Hence, LSM was able to add a single security field to this common structure and a single security field to the structure for individual messages. This is discussed further in Section 16.1.

5.5.2. Leveraging `ipcperms`

As discussed in Section 5.1.5, LSM inserts a hook into the existing Linux functions for permission checking, including the `ipcperms` function for checking access to IPC objects. The LSM-based SELinux security module leverages this hook to perform a parallel check for each existing Linux IPC permission check. However, since the SELinux IPC permissions are much finer-grained than the Linux concepts of read or write access to IPC objects, new `unix_read` and `unix_write` permissions were defined to correspond with the Linux permissions. These new permissions are checked by the hook called by `ipcperms`, and the finer-grained SELinux permissions are checked by the other IPC hooks. Hence, IPC operations require the `unix_read` or `unix_write` permission and the appropriate finer-grained permission. The coarse-grained permission checks could be omitted, but they are present to ensure that all IPC accesses are controlled by SELinux. These checks are discussed in Section 16.2.2.

5.6. Miscellaneous Changes

In addition to the changes described above, the LSM-based SELinux security module had to reimplement the approach for controlling the `sysctl` call. It also added new controls for some system operations that were not specifically addressed in the original SELinux kernel patch, such as `syslog`, which were formerly controlled only via the coarse-grained `capable` controls. Fine-grained controls over netlink operations were also introduced as part of the 2.6 SELinux. These controls are discussed in Section 19.

6. Internal Architecture

This section provides an overview of the SELinux security module internal architecture. The module code is located within the `security/selinux` subdirectory of the kernel tree. All subsequent pathnames in this section are relative to this subdirectory, unless otherwise noted. The module consists of six major components: the security server, the access vector cache (AVC), the network interface table, the netlink event notification code, the selinuxfs pseudo filesystem, and the hook function implementations.

The security server provides general interfaces for obtaining security policy decisions, enabling the rest of the module to remain independent of the specific security policies used. These interfaces are defined in the `include/security.h` header file under the SELinux module directory. The specific implementation of the security server can be changed or completely replaced without requiring any changes to the rest of the module. The example security server provided with SELinux implements a combination of Role-Based Access Control (RBAC), a generalization of Type Enforcement (TE), and optionally Multi-Level Security (MLS). The RBAC and TE policies are highly configurable and can be used to meet many different security objectives. The example security server code can be found in the `ss` subdirectory.

The AVC provides caching of access decision computations obtained from the security server to minimize the performance overhead of the SELinux security mechanisms. It provides interfaces to the hook functions for efficiently checking permissions and it provides interfaces to the security server for managing the cache. The AVC interfaces to the hook functions are defined in the `include/avc.h` header file, and the AVC interfaces to the security server are defined in the `include/avc_ss.h` header file. The AVC code can be found in the `avc.c` file.

The network interface table maps network devices to security contexts. Maintaining a separate table is necessary because the LSM network device security field was rejected. Network devices are added to the table when they are first looked up by the hook functions, and are removed from the table when the device is configured down or the policy is reloaded. The network interface table provides an interface, defined in `include/netif.h`, to the hook functions for looking up and obtaining the SIDs associated with a network device. Callback functions are registered for device configuration changes and policy reloads. The network interface table code can be found in the `netif.c` file.

The netlink event notification code allows the SELinux module to notify processes when the policy has been reloaded and when the enforcing status is changed. These notifications are used by the userspace AVC (part of libselinux) to keep its state consistent with the kernel. The userspace AVC is used by userspace policy enforcers such as security-enhanced X and security-enhanced dbus. The netlink event notification code can be found in the `netlink.c` file.

The selinuxfs pseudo filesystem exports the security server policy API to processes. The original SELinux kernel API was decomposed into three orthogonal components (process attributes, file attributes, policy API) as part of the redesign for inclusion in mainline Linux 2.6, and selinuxfs provides the underlying support for the policy API calls. All three components of the new kernel API are encapsulated by the higher level libselinux API. The selinuxfs code can be found in the `selinuxfs.c` file.

The hook function implementations manage the security information associated with kernel objects and perform the SELinux access controls for each kernel operation. The hook functions call the security server and access vector cache to obtain security policy decisions and apply those decisions to label and control kernel objects. The hook functions also call the filesystem extended attribute code to obtain and set security contexts on files. The code for these hook functions is located in the file `hooks.c`, and the data structures for the security information associated with the kernel objects are defined in the file `include/objsec.h`.

Abstractly, the hook function and data structure contents can be viewed as the same processing and data that was directly inserted into the kernel code and data structures by the original SELinux patch. However, in practice, it was often necessary to revisit the approach used by the original SELinux patch since the LSM hook locations did not always correspond to the insertion points of the original SELinux patch. In part, this was because the LSM project placed a heavier emphasis on minimizing hooks, especially outside of the core kernel code. For example, the lack of any filesystem-specific hooks required a different approach for labeling both persistent filesystems like ext3 and pseudo filesystems like procfs. Similarly, since LSM leverages the existing NetFilter framework to support hooking on many network operations, the implementation of the SELinux network access controls was changed. Nonetheless, it was possible to provide the desired security semantics with the LSM hooks.

7. Initialization

This section describes the initialization code for the SELinux security module. SELinux initialization begins with the `selinux_init` function, which is registered as a security initcall and called early in the kernel initialization sequence. Certain aspects of SELinux initialization must be deferred until later in the kernel initialization sequence and are handled by ordinary initcalls, including `selinux_nf_ip_init`, `sel_netif_init`, `selnl_init`, and `init_sel_fs`. SELinux initialization is not fully completed until after the initial policy is loaded by /sbin/init, at which point the `selinux_complete_init` function is called. Each of these functions is described below.

7.1. selinux_init

This function, located in the `hooks.c` file, handles early initialization for the SELinux module. The function starts by setting the security state for the initial task. It then calls the `avc_init` function to initialize the AVC. This initialization must be done prior to any permission checking calls to the AVC. The function then sets the secondary security module to the original security module, typically the dummy module, to support stacking with the dummy or capabilities modules. This is discussed further in Section 8. Finally, this function calls the LSM `register_security` function to register the SELinux security module as the primary security module for LSM.

7.2. selinux_nf_ip_init

This function, also located in the `hooks.c` file, handles initialization of the SELinux NetFilter hooks used to apply permission checks on outgoing packets. This function calls the `nf_register_hook` function to register the SELinux post-routing hook functions with the Netfilter framework for ipv4 and ipv6. These hook functions are discussed further in Section 18.

7.3. sel_netif_init

This function, located in the `netif.h` file, handles initialization of the SELinux network interface table that is used to look up the SIDs of network devices. This function begins by initializing the SELinux network interface hash table. It then registers a network device notifier so that it can flush entries for devices that are downed. Finally, it also registers an AVC callback so that it can flush the entire table upon a policy reload.

7.4. selnl_init

This function, located in the `netlink.c` file, handles initialization of the kernel SELinux netlink socket used to send notifications of setenforce and policy load events to userspace. The function creates the netlink socket and sets it to allow non-root processes to receive notifications so that userspace object managers are not required to run as root.

7.5. init_sel_fs

This function, located in the `selinuxfs.c` file, handles initialization of the selinuxfs pseudo filesystem. It registers the selinuxfs filesystem type and creates a private kernel mount of selinuxfs. This results in a populated selinuxfs filesystem and sets up the special null device node used by SELinux when it closes unauthorized files upon a context-changing execve.

7.6. selinux_complete_init

This function, located in the `hooks.c` file, completes the initialization of SELinux after the initial policy has been loaded by /sbin/init. It traverses a list of superblocks that were initialized prior to the initial policy load and invokes the `superblock_doinit` function on each of them. The `superblock_doinit` function proceeds to set up the security structure for each of these superblocks. It also invokes the `inode_doinit` function to set up the security structure for any existing inodes associated with the superblock.

8. Stacking with Other Modules

This section describes the current support for stacking SELinux with other security modules. LSM provides only minimal support for stacking security modules, providing hooks for this purpose but deferring the details of how stacking is handled to the primary security module. Work is ongoing to enhance the stacking support of LSM as well as to provide a generic stacker module; see the linux-security-module mailing list for more information. At present, the SELinux security module only functions as a primary security module and provides minimal support for using either the dummy security module (traditional superuser logic) or the capabilities security module as a secondary security module. This allows SELinux to be combined with either the traditional superuser logic or with the Linux capabilities logic.

As mentioned in Section 7, the `selinux_init` function initializes the secondary security module to the dummy security module, which is always resident in the kernel, prior to registering the SELinux security module. This allows the SELinux hook functions to safely call the secondary hook functions. The `selinux_register_security` hook function sets the secondary security module to a different module, such as the capabilities module. The `selinux_unregister_security` hook function restores the secondary security module to the dummy security module.

The dummy and capabilities security modules only implement a very small subset of the hook functions. Hence, at present, the SELinux security module only calls the secondary security module for this small set of hooks, along with a few other hooks that were added upon request for other security module writers. Since some of the dummy and capability hook functions are implemented in terms of the `capable` function, stacking the `capable` hook is sufficient to cover several hooks without needing to explicitly call the secondary module from the individual hook. However, in most cases, there would be no harm other than performance in always calling the secondary security module.

There are a few exceptions where calling the secondary module would pose a problem for SELinux. The capability `inode_removexattr` and `inode_setxattr` functions require `CAP_SYS_ADMIN` for all attributes in the security namespace, whereas SELinux applies its own finer-grained checking to the `security.selinux` attribute, so SELinux must completely override the secondary module for these hooks. The capability `vm_enough_memory` function would perform duplicate vm accounting if SELinux called it, so SELinux must override it as well. The capability `netlink_send` function sets the effective capability bitmap in the control buffer for the netlink message from the current process' credentials, whereas SELinux overrides this function with one that also applies SELinux permission checking.

The dummy and capabilities security modules are easy to stack with SELinux because they do not use the security fields LSM added to the kernel data structures. Stacking the SELinux module with any module that does use these fields will require the definition of a common security object header with a module identifier and a link for chaining multiple security objects on a single security field. Work is ongoing in this area on the linux-security-module mailing list.

9. SELinux API

This section discusses how the SELinux API was implemented in the SELinux security module. Prior to merging SELinux into the mainline Linux 2.6 kernel, the SELinux API was implemented using a security system call multiplexer provided by the LSM kernel patch. However, this security system call was removed by the kernel developers during the Linux 2.5 development series, thereby requiring a reworking of the SELinux API to gain acceptance into mainline Linux. The SELinux API has been refactored into three components: a new `/proc/pid/attr` API for process attributes, the existing xattr API for file attributes (using a new security namespace), and a selinuxfs pseudo filesystem for the security policy API. Support for the SELinux extensions for System V and socket IPC will be investigated in the future. All three components of the new SELinux API are encapsulated by the libselinux library.

As part of the redesign of the SELinux API for Linux 2.6, SIDs were removed from the API, and only security contexts are now passed by the API calls. This approach provided a better fit with both the /proc-based interface and the xattr-based interface, as well as providing a better fit with the needs of most applications. It also allows for future implementation of kernel tracking of SID usage and safe reclamation of unused SIDs. For applications that would benefit from a SID abstraction, e.g. userspace object managers such as dbusd, nscd, or X, a userspace SID table was implemented in libselinux along with the userspace AVC.

Prior to merging SELinux into mainline Linux 2.6, extended system calls such as `execve_secure`, `open_secure` and `stat_secure` were implemented by SELinux to allow security information to be provided or returned by a call. In the original SELinux kernel patch, these calls were implemented internally by extending the internal kernel functions to optionally pass and process SID parameters. This approach was viewed as enhancing the Linux API to incorporate security as a first class notion, retaining the original calls for compatibility but re-implementing them internally by passing default SIDs to the extended internal functions. Later, in the LSM-based SELinux, to reduce the invasiveness of SELinux, the calls were re-implementing by passing SIDs via fields in the task security structure to and from the security hook functions. However, this approach of a parallel set of extended calls for existing system calls was not welcomed by the kernel developers.

Based on kernel developer feedback, the extended system calls of the original SELinux API were replaced with separate set-attribute calls that precede an ordinary call. For example, a call to `execve_secure` is replaced with a call to `setexeccon` to set the desired exec security context followed by a call to `execve` to perform the exec. Similarly, a call to `open_secure` or `mkdir_secure` is replaced with a call to `setfscreatecon` to set the desired filesystem creation context followed by a call to `open` or `mkdir` to perform the file creation. The exec context and fscreate context are attributes of the process like the umask; if set, they are applied to all subsequent execve or file creation calls until they are explicitly cleared via `setexeccon` or `setfscreatecon` calls with a NULL argument or they are automatically cleared after an execve (i.e. reset upon each new program execution). Within libselinux, these calls are implemented via writes to `/proc/self/attr/exec` and `/proc/self/attr/fscreate` nodes. Note that a process can only set its own exec and fscreate contexts.

The new SELinux API allows for simplification of applications that merely wish to set a single exec or fscreate context and have it applied for any subsequent execve or file creation call, since the setting of the context can be performed up front and the SID/context does not need to be passed around by the application functions. The API also avoids the need to extend various library functions (e.g. execl*, execv*, popen/pclose, fopen) that internally perform execve or file creation calls, since the caller can simply set the exec or fscreate context prior to making the ordinary library function call and have the context automatically applied when the execve or file creation call is made by the library function.

However, this API does require additional care to be taken by library functions to save and restore the exec or fscreate contexts if they need to set them temporarily for their own processing (e.g. to preserve the security context on `/etc/shadow`) and by signal handlers to save, reset, and restore these contexts if the signal handler calls execve or a file creation call. The setting of the /proc/pid/attr attributes is not supported for multi-threaded processes.

10. Helper Functions for Hook Functions

The SELinux security module provides a set of helper functions that are used extensively by the SELinux hook implementations. This section provides an overview of these helper functions. More detailed descriptions of individual helper functions are provided in the appropriate hooks section.

10.1. Primitive Allocation Helper Functions

For most SELinux security data structures defined in `include/objsec.h`, the SELinux module provides a primitive `alloc_security` and `free_security` helper function, e.g. `task_alloc_security` and `task_free_security`. These helper functions are used by the `alloc_security` and `free_security` hook functions, which may contain additional processing beyond the basic initialization performed by these helpers.

Each primitive `alloc_security` helper function allocates a security structure of the appropriate type, sets a back pointer to the kernel data structure, initializes the security information, and sets the object security field to refer to this new security structure. Each primitive `free_security` helper function clears the security field and frees the security structure.

10.2. Initialization Helper Functions

The SELinux security module defines initialization helper functions for certain security structures (e.g. `inode_doinit`, `superblock_doinit`, etc). These initialization helper functions are called by certain SELinux hook functions, and are discussed further in later sections.

10.3. Permission Checking Helper Functions

A set of helper functions on kernel objects and permissions are provided that dereference the security fields, set up auxiliary audit data, and then invoke the access vector cache (AVC) to perform the permission check with the right set of parameters. These helper functions simplify the code for many of the hook functions that perform permission checks. A few examples of these functions include `task_has_perm`, `inode_has_perm`, and `may_create`.

Although these helper functions can be convenient, hook functions are free to directly call the AVC to perform permission checks. This is done in a couple of cases. First, some permission checks involve a security identifier (SID) that is not associated with a kernel object, e.g. a SID specified by an application using the SELinux API calls or a SID obtained from the security server for an object that is about to be created. Second, some operations require multiple permission checks to be performed that are based on some of the same SIDs. In the latter case, using the helper functions would cause redundant processing in order to extract the same SIDs multiple times.

11. Task Hook Functions

The SELinux task hook function implementations manage the security fields of task_struct structures and perform access control for task operations. This section describes these hooks and their helper functions.

11.1. Managing Task Security Fields

11.1.1. Task Security Structure

The task_security_struct structure contains security information for tasks. This structure is defined as follows:

```
struct task_security_struct {
        struct task_struct *task;
        u32 osid;
        u32 sid;
        u32 exec_sid;
        u32 create_sid;
        u32 ptrace_sid;
};
```

Table 1. task_security_struct

Field	Description
task	Back pointer to the associated task_struct structure.
osid	SID prior to the last execve.
sid	current SID for the task.
exec_sid	SID for the task upon the next execve call.
create_sid	SID for files created by the task.

11.1.2. task_alloc_security and task_free_security

The `task_alloc_security` and `task_free_security` helper functions are the primitive allocation functions for task security structures. The `selinux_task_alloc_security` hook function calls `task_alloc_security` for the new task and then copies the SID fields from the current task into the new task. The `selinux_task_free_security` hook function simply calls the corresponding helper function.

11.1.3. selinux_task_reparent_to_init

This hook function is called by the kernel `reparent_to_init` function to set the security attributes for a kernel task. This hook function first calls the secondary security module to support Linux capabilities. It then sets the SID of the task to the `kernel` initial SID.

11.1.4. selinux_task_post_setuid

This hook function is called after a setuid operation has successfully completed. Since the SELinux module does not use the Linux identity attributes, this hook function does not perform any SELinux processing. However, it does call the secondary security module to support Linux capabilities.

11.1.5. selinux_task_to_inode

This hook function is called by the procfs pseudo filesystem to set the security state for the /proc/pid inodes associated with a task. This function sets the inode SID from the task SID and marks the inode security structure as initialized.

11.1.6. selinux_getprocattr

This hook function is called by the procfs pseudo filesystem to get a process security attribute value from the security module upon an attempt to read a node under the /proc/pid/attr directory. The hook function begins by checking `getattr` permission if the target task differs from the current task. It then extracts the appropriate SID from the task security structure. If the corresponding SID has not been set (e.g. if no explicit exec SID has been set and the task is using the default policy behavior), then the hook returns a zero length. Otherwise, the hook function calls `security_sid_to_context` to obtain the security context associated with the SID, copies the context to the provided kernel buffer (if it is large enough), and returns its length.

11.1.7. selinux_setprocattr

This hook function is called by the procfs pseudo filesystem to set a process security attribute value from the security module upon an attempt to write a node under the /proc/pid/attr directory. The hook function begins by checking whether the target task differs from the current task, returning an error in that case to prevent setting of a task's security attributes by another task. The function then applies a permission check between the current task and the target task (always a self relationship due to the prior restriction) based on the particular attribute being set. If a context was written to the node (as opposed to writing a zero length buffer to reset an exec or fscreate SID to the default policy behavior), then the function calls `security_context_to_sid` to convert it to a SID.

If the attribute is the exec or fscreate context, then the function proceeds to set the corresponding SID in the task security structure. For these attributes, further permission checks based on the specified security context are not performed until the execve or file creation operation occurs, at which point that operation may fail due to a lack of permission. This is partly a legacy of the original API, where extended system calls specified the SID for the operation as part of the operation call rather than separately setting a process attribute in advance. While it would be possible to duplicate some of this checking within the `selinux_setprocattr` hook function (e.g. process transition permission check), the hook function lacks the full context of the execve or file creation operation, e.g. the entrypoint program for execve and the parent directory, filesystem, and specific file type for file creation.

If the attribute is the current context (i.e. a dynamic context transition), then the hook function verifies that there are no other threads in the process, checks `dyntransition` permission between the old and new task SIDs, and if the process is being traced, checks `ptrace` permission between the tracer SID and the new SID. If all checks pass, then the task SID is set to the new value.

11.2. Controlling Task Operations

11.2.1. Helper Functions for Checking Task Permissions

Several helper functions are provided for performing task permission checks. These functions and their permission checks are summarized in Table 2. The `task_has_perm` function checks whether a task has a particular permission to another task. The `task_has_capability` function checks whether a task has permission to use a particular Linux capability. The `task_has_system` function checks whether a task has one of the permissions in the `system` security class. This security class is used for permissions that control system operations when there is no existing capability check or the capability check is too coarse-grained. The `task_has_security` function checks whether a task has permission to use one of the selinuxfs APIs.

Table 2. Task Helper Function Permission Checks

Function	Source	Target	Permission(s)
task_has_perm	SourceTask	TargetTask	ProcessPermission
task_has_capability	Task	Task	CapabilityPermission
task_has_system	Task	`Kernel`	SystemPermission
task_has_security	Task	`Security`	SecurityPermission

Except for `task_has_perm`, these permission checks are simply based on a single task, so the target SID is unnecessary. In the case of `task_has_capability`, the task's SID is passed for both the source and target SIDs. For `task_has_system` and `task_has_security`, a distinct initial SID is used for the target SID.

11.2.2. Hook Functions for Controlling Task Operations

The task hook functions that perform access control and their permission checks are summarized in Table 3. These functions call the `task_has_perm` helper function.

Table 3. Task Hook Function Permission Checks

Hook	Source	Target	Permission(s)
selinux_task_create	Current	Current	fork
selinux_task_setpgid	Current	TargetTask	setpgid
selinux_task_getpgid	Current	TargetTask	getpgid
selinux_task_getsid	Current	TargetTask	getsession
selinux_task_getscheduler	Current	TargetTask	getsched
selinux_task_setscheduler selinux_task_setnice	Current	TargetTask	setsched
selinux_task_kill	Current	TargetTask	signull sigchld sigkill sigstop signal
selinux_task_wait	ChildTask	Current	sigchld sigkill sigstop signal
selinux_task_setrlimit	Current	Current	setrlimit
selinux_ptrace	Parent	Child	ptrace

Only three of these hook functions require further explanation. The `selinux_task_kill` hook function checks a permission between the current task and the target task based on the signal being sent. The `selinux_task_wait` checks a permission between the child task and the current task based on the exit signal set for the child task. This allows control over the ability of a process to reap a child process of a different SID. In both hooks, the `SIGKILL` and `SIGSTOP` signals have their own distinct permissions because neither of these two signals can be blocked. The `SIGCHLD` signal has its own distinct permission because it is commonly sent from child processes to parent processes. The `signull` permission is checked if a 0 signal is passed to kill, as this merely represents an existence test, not an actual signal delivery. For all other signals, the generic `signal` permission is used.

The `selinux_task_rlimit` hook checks `setrlimit` permission if a hard limit is being changed so that the hard limit can later be used as a safe reset point for the soft limit upon context transitions. See the section on `selinux_bprm_apply_creds` for further discussion of the resource limit inheritance control.

In addition to checking ptrace permission, the `selinux_ptrace` hook also sets the tracer SID in the child task's security structure for later use by `selinux_bprm_apply_creds` and `selinux_setprocattr`. See Section 12.1.4 and Section 11.1.7 for further discussion.

Several of the task hook functions for controlling operations are not used by the SELinux security module. These hook functions are:

- selinux_task_setuid
- selinux_task_setgid
- selinux_task_setgroups
- selinux_task_prctl

Since SELinux does not depend on the Linux identity attributes, and since these operations can only affect the current process, SELinux does not need to control these operations. Privileged aspects of these operations are already controlled via the `selinux_capable` hook function. However, it may be desirable in the future to add SELinux permissions to control these operations, e.g. to confine Linux identity changes at finer granularity.

12. Program Loading Hook Functions

The SELinux binprm hook function implementations manage the security fields of linux_binprm structures and perform access control for program loading operations. This section describes these hooks and their helper functions.

12.1. Managing Binprm Security Fields

12.1.1. Binprm Security Structure

The bprm_security_struct structure contains security information for program loading. This structure is defined as follows:

```
struct bprm_security_struct {
        struct linux_binprm *bprm;
        u32 sid;
        unsigned char set;
        char unsafe;
};
```

Table 4. task_security_struct

Field	Description
bprm	Back pointer to the associated linux_binprm structure.
sid	SID for the transformed process.
set	Flag indicating whether sid has been set.
unsafe	Flag indicating whether an unsafe transition was attempted.

12.1.2. selinux_bprm_alloc_security and selinux_bprm_free_security

The `selinux_bprm_alloc_security` and `selinux_bprm_free_security` hook functions directly perform allocation and freeing of the linux_binprm security structures rather than using primitive helper functions. However, they perform the same basic processing as described in Section 10.1. In earlier versions of the SELinux module, there was no bprm security structure and these functions did nothing for SELinux, as only the new SID for the transformed process was needed and it was stored directly in the linux_binprm security field.

12.1.3. selinux_bprm_set_security

The `selinux_bprm_set_security` hook function is called while loading a new program to fill in the linux_binprm security field and optionally to check permissions. This hook function may be called multiple times during a single execve, e.g. for interpreted scripts. This hook function first calls the secondary security module to support Linux capabilities. If the set flag in the bprm security structure has already been set by a prior call to this hook, this hook merely returns without further processing. This allows security transitions to occur on scripts if permitted by the policy. Naturally, such transitions should only occur when the caller is more trusted than the new domain, as script invocation is subject to an inherent race and scripts are highly susceptible to influence by their caller. However, SELinux does allow transitions on scripts subject to policy, e.g. to support shedding of permissions upon script invocation where the caller is trusted.

By default, this hook function sets the SID in the bprm security structure to the SID of the current task. It also clears any file creation SID set earlier by the task to ensure that the new program starts with a clean initial state. This function checks the current task's security structure to see if the task specified an exec SID for the next execve call. If so, then this exec SID is used and cleared. Otherwise, the security server is consulted using the `security_transition_sid` interface to see whether the SID should change based on the current SID of the task and the SID of the program. If the filesystem is mounted `nosuid`, then any exec SID set previously or transition SID obtained from the security server is ignored, and the task SID is left unchanged.

This hook function then performs different permission checks depending on whether the SID of the task is changing. The permission checks for each case are described below. The file `execute` permission check is performed by the `selinux_inode_permission` hook during `open_exec` processing, so it is not listed here.

The file `execute_no_trans` permission is checked when a task would remain in the same SID upon executing a program, as shown in Table 5. This permission check ensures that a task is allowed to execute a given program without changing its security attributes. For example, although the login process can execute a user shell, it should always change its SID at the same time, so it does not need this permission to the shell program.

Table 5. Permission Checks if Task SID is not changing on exec

Source	Target	Permission(s)
Current	ProgramFile	execute_no_trans

The process `transition` permission and the file `entrypoint` permission are checked when the SID of a task changes. The `transition` permission check ensures that the old SID is allowed to transition to the new SID. The `entrypoint` permission check ensures that the new SID can only be entered by executing particular programs. Such programs are referred to as entrypoint programs for the SID. These permission checks are shown in Table 6. If all permission checks for a transition pass, then any unsafe personality bits are cleared and the new SID is saved in the bprm security structure for use by `selinux_bprm_apply_creds`.

Table 6. Permission Checks if Task SID is changing on exec

Source	Target	Permission(s)
Current	NewTaskSID	transition
NewTaskSID	ProgramFile	entrypoint

12.1.4. selinux_bprm_apply_creds

The `selinux_bprm_apply_creds` hook function is called to set the new security attributes for the transformed process upon an execve after checking for certain unsafe conditions with the task lock held. This hook function first calls the secondary security module to support Linux capabilities. This hook then extracts the new task SID from the bprm security structure, copies the current SID of the task into the old SID field of the task security structure, and clears the unsafe flag in the bprm security structure. If the new SID is the same as the old SID, then nothing further is done by this hook.

Two additional permission checks may occur when the SID of the task is changing. If the task was created via `clone` and has shared state, then the `share` permission is checked between the old and new SIDs. If the task is being traced, then the `ptrace` permission is checked between the tracer task (saved in the ptrace_sid field of the current task's security structure earlier upon ptrace_attach) and the new SID. The permission checks are shown in Table 7. If either of these permission checks fail, then the task SID is left unchanged, the unsafe flag is set in the bprm security structure for later use by the `selinux_bprm_post_apply_creds` hook, and the hook immediately returns. If all permissions are granted, this hook function changes the SID of the task to the new SID and returns.

Table 7. Permission Checks if Task SID is changing on exec

Source	Target	Permission(s)
TracerTask	NewTaskSID	ptrace
Current	NewTaskSID	share

12.1.5. selinux_bprm_post_apply_creds

This hook function is called after the `selinux_bprm_apply_creds` without the task lock held to support further processing that cannot be done safely under the task lock or that does not require it to be held. The hook function first checks whether the unsafe flag was set in the bprm security structure, and if so, it forces a SIGKILL to the task to terminate it and returns immediately. This function then checks whether the task SID has changed, and if not, it returns immediately, as no further processing is required.

Otherwise, this hook function proceeds to call the `flush_unauthorized_files` helper function to revoke access to the controlling tty if the task is no longer allowed to access it and to close any file descriptors to which the task should no longer have access. After revalidating access to the controlling tty and revoking it if necessary, the helper function calls `file_has_perm` on each open file with requested permissions that correspond to the file mode and flags, and closes the open file if these permissions are not granted under the new SID. The `file_has_perm` function is described in Section 15.2.1. To avoid inducing errors in applications that expect certain descriptors to be open, this helper function also re-opens any descriptors it closes to refer to a null device node that was set up in selinuxfs during initialization. The helper function then returns.

The `selinux_bprm_post_apply_creds` hook function then applies two inheritance checks between the old and new SIDs, one to control the ability to inherit signal-related state and one to control the ability to inherit resource limits from the old SID. These checks are intended to protect the program in the new SID against certain forms of influence by the caller unless authorized by policy. If the `siginh` permission is denied, then any itimers are cleared to avoid subsequent signal generation, pending signals are flushed and unblocked, and all signal handlers are reset to the default. If the `rlimitinh` permission is denied, then all soft resource limits are reset to the lower of the current task's hard limit and the initial task's soft limit. This control relies on the proper control of the `setrlimit` permission to prevent untrusted processes from lowering hard limits as well. The inclusion of the initial task's soft limits into the computation is to avoid resetting soft limits higher than the default soft limit for cases where the default is lower than the hard limit, e.g. `RLIMIT_CORE` or `RLIMIT_STACK`. These two inheritance checks are shown in Table 8.

Table 8. Inheritance Permission Checks if Task SID is changing on exec

Source	Target	Permission(s)
OldTaskSID	NewTaskSID	siginh
OldTaskSID	NewTaskSID	rlimitinh

Finally, this hook function wakes up the parent task if it is waiting on this task. This allows the `selinux_task_wait` hook to recheck whether the parent task is allowed to wait on the task under its new SID and to handle a denial appropriately.

12.1.6. selinux_bprm_secureexec

This hook function is called after `selinux_bprm_post_apply_creds` to determine whether the AT_SECURE flag in the ELF auxiliary table should be set to cause glibc to enable its secure mode in order to sanitize the environment and other state to protect the new program against certain forms of influence by the caller. If the task SID has changed, then this hook function checks `noatsecure` permission between the old and new task SIDs. If this permission is denied, the hook function will set the AT_SECURE flag so that libc secure mode will be enabled. If the permission is allowed, the hook function calls the secondary security module to allow it to set the flag, e.g. if there is a change in uid, gid or capabilities. Otherwise, the flag will not be set and libc secure mode will not be enabled.

13. Superblock Hook Functions

The SELinux superblock hook function implementations manage the security fields of super_block structures and perform access control for filesystem operations. This section begins by describing the superblock hook functions for managing the security fields. It then discusses the superblock hook functions for performing access control.

13.1. Managing Superblock Security Fields

13.1.1. Superblock Security Structure

The superblock_security_struct structure contains security information for superblock objects. This structure is defined as follows:

```
struct superblock_security_struct {
        struct super_block *sb;
        struct list_head list;
        u32 sid;
        u32 def_sid;
        unsigned int behavior;
        unsigned char initialized;
        unsigned char proc;
        struct semaphore sem;
        struct list_head isec_head;
        spinlock_t isec_lock;
};
```

Table 9. superblock_security_struct

Field	Description
sb	Back pointer to the associated superblock.
list	Link into list of superblock security structures setup prior to initial policy load.
sid	SID for the file system.
def_sid	default SID for labeling
behavior	Labeling behavior to apply to inodes.
initialized	Flag indicating whether the security structure has been initialized.
sem	Semaphore used to synchronize initialization.
isec_head	List of inode security structures setup prior to superblock security initialization.

Field	Description
isec_lock	Lock for list of inode security structures.

13.1.2. superblock_alloc_security and superblock_free_security

The superblock_alloc_security and superblock_free_security helper functions are the primitive allocation functions for super_block security structures. The selinux_sb_alloc_security and selinux_sb_free_security hook functions call these helper functions.

13.1.3. superblock_doinit

This helper function performs initialization for superblock security structures. It is normally called by the selinux_sb_kern_mount hook function. However, since this helper function cannot perform full initialization until after the initial policy load, it is also called by the selinux_complete_init function to retroactively complete initialization of superblocks setup prior to the initial policy load.

The superblock_doinit function begins by taking the semaphore to synchronize with any other attempts to initialize the superblock security data and then checks whether the superblock security structure has already been marked initialized. If so, the function returns after releasing the semaphore. Otherwise, it checks whether the initial policy load has completed. If not, then the function adds the superblock security structure to a global list for deferred processing by selinux_complete_init, releases the semaphore and returns.

If the initial policy load has completed, then superblock_doinit calls the security server's security_fs_use interface to determine the labeling behavior for the inodes associated with the superblock and to obtain a SID for the superblock itself. It then calls the try_context_mount to handle any mount context options, which can override the labeling behavior and superblock SID returned by security_fs_use. Context mount support is discussed further in Section 13.1.5.

If the desired labeling behavior is to use extended attributes (xattr) superblock_doinit verifies that the filesystem supports xattr and the security namespace, returning an error otherwise. If the superblock is for the proc pseudo filesystem, the function sets the proc flag in the superblock security structure for special handling upon inode initialization. The function then marks the superblock security structure as initialized.

Next, superblock_doinit calls inode_doinit_with_dentry on the root inode to initialize its security structure. The function likewise makes calls to initialize the inode security structures for any inodes that were setup prior to superblock security initialization (e.g. prior to initial policy load or during get_sb by a filesystem that directly populates itself). Finally, the function releases the semaphore and returns.

13.1.4. selinux_sb_copy_data

The selinux_sb_copy_data function is called to allow mount option data to be copied prior to parsing by the filesystem, so that the security module can extract security-specific mount options cleanly, especially since the filesystem code may modify the data while processing. The hook function also allows the security-specific options to be stripped from the mount data so that the filesystem code does not need to be aware of them at all. If the filesystem type uses binary mount data, then this hook function simply copies the binary data into the page for security data for later processing. Otherwise, this hook function parses the mount option string and extracts any mount context options for later use. There are three kinds of mount context options: context, fscontext, and defcontext.

13.1.5. try_context_mount

The `try_context_mount` function is called to handle any context mount options extracted earlier by `selinux_sb_copy_data`. If the filesystem type uses binary mount data, then the function will extract context options if using a version of the binary mount data that includes them. At present, only NFS has support for such options, and it only supports one of the context mount options (context=). For string mount data, the function processes each context-related option, checking for invalid combinations. The fscontext and defcontext options may be used together as well as individually, but no other combination of options is allowed. If any context option is specified, the function applies several permission checks among the task SID, the original superblock SID, and the SID for the provided context to verify that the use of the option is authorized.

For the context or fscontext options, the superblock SID is set to the SID for the provided context. The context option further changes the labeling behavior to mountpoint labeling, which means that all inodes in the filesystem are treated as having the provided context as well, and the xattr API is not supported for the inodes in the filesystem even if the filesystem type itself supports xattrs. In contrast, the fscontext option only sets the superblock SID and leaves the labeling behavior unchanged. The defcontext option only sets the default SID (def_sid) for inodes in the filesystem, overriding the typical value of the file initial SID.

13.1.6. selinux_sb_kern_mount

This hook function is called to setup the superblock security structure and to check permissions for the mount of a particular superblock. It calls `superblock_doinit` to perform the initialization and then calls `superblock_has_perm` to check filesystem mount permission to the superblock.

13.2. Controlling Filesystem Operations

13.2.1. superblock_has_perm

This helper function checks whether a task has a particular permission to a filesystem. It takes the task, the super_block, the requested permissions, and optionally audit data as parameters. This function simply calls the AVC with the appropriate parameters.

13.2.2. selinux_sb_statfs

This hook function is called to check permission when obtaining filesystem attributes. It checks `getattr` permission between the current task and the filesystem.

13.2.3. selinux_mount

This hook function is called to check permission when mounting a filesystem prior to the actual reading of the superblock. If the filesystem is being remounted (i.e. the mount flags are being changed), then this function checks `remount` permission between the current task and the filesystem. Otherwise, this function checks `mounton` permission between the current task and the mountpoint directory.

13.2.4. selinux_umount

This hook function is called to check permission when unmounting a filesystem. This function checks `unmount` permission between the current task and the filesystem.

13.2.5. selinux_quotactl

The `selinux_quotactl` hook function checks that the current task has permission to perform a given quota control command on a filesystem. If no filesystem was specified (i.e. a `Q_SYNC` or `Q_GETSTATS` command), then the hook simply returns success, since these operations require no control. Otherwise, one of the `quotamod` or `quotaget` permissions is checked between the current task and the filesystem, depending on whether the command sets information or merely gets information related to quotas.

13.2.6. Summary of Filesystem Permission Checks

The permission checks for the super_block hooks are summarized in Table 10.

Table 10. Filesystem Permission Checks

Hook	Source	Target	Permission(s)
selinux_sb_statfs	Current	Filesystem	getattr
selinux_mount	Current Current	MountDirectory Filesystem	mounton remount
selinux_sb_kern_mount	Current	Filesystem	mount
selinux_umount	Current	Filesystem	unmount
selinux_quotactl	Current	Filesystem	quotamod quotaget

14. Inode Hook Functions

The SELinux inode hook function implementations manage the security fields of inode structures and perform access control for inode operations. Since inodes are used to represent pipes, files, and sockets, the hook functions must handle each of these abstractions. Furthermore, these hooks must handle multiple filesystem types, including both conventional disk-based filesystems like ext3 and pseudo filesystems like proc and tmpfs. This section begins by describing the inode hook functions for managing the security fields. It then discusses the inode hook functions for performing access control.

14.1. Managing Inode Security Fields

14.1.1. Inode Security Structure

The inode_security_struct structure contains security information for inodes. This structure is defined as follows:

```
struct inode_security_struct {
        struct inode *inode;
        struct list_head list;
        u32 task_sid;
        u32 sid;
        u16 sclass;
        unsigned char initialized;
        struct semaphore sem;
};
```

Table 11. inode_security_struct

Field	Description
inode	Back pointer to the associated inode.
list	Link into list of inode security structures setup prior to superblock security initialization.
task_sid	SID of the task that allocated this inode.
sid	SID of this inode.
sclass	Security class of this inode.
initialized	Flag indicating whether the inode SID has been initialized.
sem	Semaphore for synchronizing initialization.

14.1.2. inode_alloc_security and inode_free_security

The `inode_alloc_security` and `inode_free_security` helper functions are the primitive allocation functions for inode security structures. In addition to the general processing for these primitive allocation functions, `inode_alloc_security` saves the SID of the allocating task in the *`task_sid`* field. The `selinux_inode_alloc_security` and `selinux_inode_free_security` hook functions merely calls these helper functions.

14.1.3. inode_doinit, selinux_d_instantiate

The `inode_doinit_with_dentry` helper function performs initialization for inode security structures.

It is normally called for file inodes by the `selinux_d_instantiate` hook function. However, since this helper function cannot perform full initialization until after the superblock security initialization is complete for the associated superblock, it is also called by the `superblock_doinit` function to retroactively complete initialization of inodes setup prior to the superblock security initialization. This includes both inodes setup prior to the initial policy load and any inodes directly populated by the filesystem code during get_sb processing.

This helper function begins by checking the initialized flag to see whether the inode SID has already been initialized and, if so, jumps to the code for setting the inode security class. Setting of the inode security class is always performed by this function if it has not been previously set to a more specific value than the initial default file class even even if the initialized flag has been previously set, as the inode mode is not always properly set at the time when an inode SID is first set. In particular, this is the case for /proc/pid inodes.

If the initialized flag has not been set, this function takes the semaphore to synchronize with any other attempts to initialize the inode SID and rechecks the initialized flag again. The function then proceeds to check whether the superblock security structure has been initialized. If not, the inode security structure is placed on the list maintained in the superblock security structure for deferred processingby `superblock_doinit` and the function returns after releasing the semaphore.

If the superblock security structure has been initialized, then this function sets the inode SID based on the defined labeling behavior for the superblock. If the labeling behavior is to use extended attributes (xattr), then this function invokes the getxattr method to fetch the context value and invokes `security_context_to_sid_default` to convert it to a SID, possibly inheriting some information from the default file SID associated with the superblock. If the inode has no xattr value, then the inode is assigned the default SID from the superblock security structure, which is either the initial file SID or a SID specified via the defcontext mount option.

If the labeling behavior is to inherit the inode SID directly from the allocating task, then the function copies the task SID from the inode security structure into its own SID field. This behavior is used for private objects such as socket and pipes.

If the labeling behavior is to compute the inode SID based on both the allocating task SID and the superblock SID, then the security servers's `security_transition_sid` function is invoked to obtain the inode SID. This behavior is used for pseudo filesystems like devpts and tmpfs where the inodes are labeled with derived types reflecting both their creator and the kind of object (e.g. a pty, a temporary file). As discussed in Section 13.1.5, the labeling behavior can be overridden via the context mount option, so tmpfs mounts can be assigned a particular security context instead, as is done for the tmpfs /dev used by udev.

For any other labeling behavior, the inode SID defaults to the superblock SID. There is a further refinement for the proc filesystem; if the inode is in the proc filesystem and is not a /proc/pid inode, then the `selinux_proc_get_sid` function is invoked to construct a pathname for the inode based on the proc_dir_entry information and then obtain a SID for that pathname via the security server's `security_genfs_sid` function. The proc_dir_entry information is used to ensure a stable and reliable name mapping, unlike the filesystem namespace itself. Note that /proc/pid inodes have their SIDs initialized separately by the `selinux_task_to_inode` hook function, as discussed in Section 11.1.5.

After setting the inode SID, the function sets the initialized flag in the inode security structure to indicate that the SID has been set. Finally, the function determines the security class for the inode and sets the corresponding field in the inode security structure if the security class has not already been set to a more specific value than the initial default file class. The check for a more specific value than the default file class is to avoid overwriting the class value set by the socket hooks for socket inodes, as this function cannot properly classify socket inodes. The `inode_mode_to_security_class` function is used to obtain the security class based on the inode mode. The mapping between inode modes and security classes is described in Table 12. If the inode does not have any of the modes listed in Table 12, then it defaults to the file security class.

Table 12. Inode Security Classes

Mode	Security Class
`S_IFREG`	file
`S_IFDIR`	dir
`S_IFLNK`	lnk_file
`S_IFFIFO`	fifo_file
`S_IFSOCK`	sock_file
`S_IFBLK`	blk_file
`S_IFCHR`	chr_file

14.1.4. selinux_inode_init_security

The `selinux_inode_init_security` hook function is called by the filesystem-specific code when creating a new file in order to obtain the security attribute to assign to the new inode and to set up the incore inode security structure for the new inode. This support allows new inodes to be atomically labeled as part of the inode creation transaction, ensuring that an inode is never visible without a security label. This hook and the corresponding filesystem suppport was introduced in Linux 2.6.14; prior kernel versions used a different set of post creation hooks invoked by the VFS layer that did not provide atomicity, allowing the new inode to be temporarily visible in an unlabeled state. Support for atomic inode labeling was only implemented for the ext2, ext3, tmpfs, and jfs filesystems in 2.6.14; similar support for other filesystems like xfs and reiserfs has not yet been implemented at the time of this writing.

This function first checks the current task's security structure to see if the task has set a fscreate SID for newly created files. If so and mountpoint labeling is not being used for the filesystem, then this SID is used. Otherwise, a SID is obtained from the security server by calling the `security_transition_sid` interface; passing in the creating task and parent directory SIDs. The `inode_security_set_sid` helper function is called to set the SID and security class in the incore inode security structure.

If the filesystem is using mountpoint labeling, then no attribute should be set on disk, so the function returns an `EOPNOTSUPP` error to the filesystem code to skip setting of the on-disk attribute. Otherwise, if the filesystem code supplied pointer arguments to receive the attribute name and value, the function generates the SELinux attribute name and the security context value for the inode and sets the arguments accordingly before returning successfully. Certain filesystems such as tmpfs do not provide pointer arguments for receiving the attribute name and value because there is no attribute representation other than the incore representation, unlike the disk-based filesystems that have on-disk attribute storage.

14.1.5. selinux_inode_post_setxattr

This hook function is called to update the inode security structure after a successful setxattr operation while the inode semaphore is still held. It first checks whether the changed attribute is the SELinux attribute; if not, it returns immediately. Otherwise, it converts the attribute value to a SID and updates the inode SID.

14.1.6. selinux_inode_getsecurity

This hook function was originally called on getxattr(2) calls on attributes in the security namespace for filesystems that did not provide native support for xattrs. It is now called (as of Linux 2.6.15) on all getxattr(2) calls on attributes in the security namespace, even when the filesystem supports xattrs, in order to allow SELinux to provide the canonical form of the security context to userspace. After checking that the requested attribute is the SELinux attribute, the function calls `security_sid_to_context` to convert the inode SID to a context and copies the context into the provided buffer.

14.1.7. selinux_inode_setsecurity

This hook function is called upon setxattr(2) calls on attributes in the security namespace for filesystems that do not provide native support for xattrs. After checking that the attribute name is the SELinux attribute, the function calls the `security_context_to_sid` to convert the provided attribute value to a SID and sets the inode SID to it.

14.1.8. selinux_inode_listsecurity

This hook function is called upon listxattr(2) calls to return the names of any security attributes supported by the security module for filesystems that do not provide native support for xattrs. It copies the name of the SELinux attribute into the provided buffer.

14.2. Controlling Inode Operations

14.2.1. inode_has_perm

This helper function checks whether a task has a particular permission to an inode. In addition to taking the task, inode, and requested permission as parameters, this function takes an optional auxiliary audit data parameter. This optional parameter allows other audit data, such as the particular dentry, to be passed for use if an audit message is generated. This function sets up an auxiliary audit data structure if one is not provided and then calls the AVC to check the requested permission to the inode.

14.2.2. dentry_has_perm

This helper function is the same as the `inode_has_perm` except that it takes a dentry as a parameter rather than an inode, and optionally takes a vfsmount parameter. This function saves the dentry and vfsmount in the audit data structure and then calls `inode_has_perm` with the appropriate parameters.

14.2.3. may_create

This helper function checks whether the current task can create a file. It takes the parent directory inode, the dentry for the new file, and the security class for the new file. This function checks the current task's security structure to see if the task has set a fscreate SID for newly files. If so and mountpoint labeling is not being used, then this SID is used. Otherwise, a SID is obtained from the security server using the `security_transition_sid` interface. The function then checks permissions as described in Table 13.

Table 13. Create Permission Checks

Source	Target	Permission(s)
Current	ParentDirectory	search, add_name
Current	File	create
File	Filesystem	associate

This helper function is called by the following inode hook functions:

- `selinux_inode_create`
- `selinux_inode_symlink`
- `selinux_inode_mkdir`
- `selinux_inode_mknod`

14.2.4. may_link

This helper function checks whether the current task can link, unlink, or rmdir a file or directory. It takes the parent directory inode, the dentry of the file, and a flag indicating the requested operation. The permission checks for these operations are shown in Table 14 and Table 15.

Table 14. Link Permission Checks

Source	Target	Permission(s)
Current	ParentDirectory	search, add_name
Current	File	link

Table 15. Unlink or Rmdir Permission Checks

Source	Target	Permission(s)
Current	ParentDirectory	search, remove_name
Current	File	unlink or rmdir

This helper function is called by the following inode hook functions:

- `selinux_inode_link`
- `selinux_inode_unlink`
- `selinux_inode_rmdir`

14.2.5. may_rename

This function checks whether the current task can rename a file or directory. It takes the inodes of the old and new parent directories, the dentry of an existing link to the file, and the new dentry for the file. This function checks the permissions described in Table 16, Table 17, and Table 18. The permissions in Table 16 are always checked. The permissions in Table 17 are only checked if the new dentry already has an existing inode (i.e. a file already exists with the new name), in which case that file will be removed by the rename. The permissions in Table 18 are only checked if the file is a directory and its parent directory is being changed by the rename.

Table 16. Basic Rename Permission Checks

Source	Target	Permission(s)
Current	OldParentDirectory	search, remove_name
Current	File	rename
Current	NewParentDirectory	search, add_name

Table 17. Additional Rename Permission Checks if NewFile Exists

Source	Target	Permission(s)
Current	NewParentDirectory	remove_name
Current	NewFile	unlink or rmdir

Table 18. Additional Rename Permission Checks if Reparenting

Source	Target	Permission(s)
Current	File	reparent

This helper function is called by the following inode hook functions:

- `selinux_inode_rename`

14.2.6. selinux_inode_permission

This hook function is called by the kernel `permission` and `exec_permission_lite` functions to check permission when accessing an inode. If the permission mask is null, then there is no permission to check and the function simply returns success. This can occur upon file existence tests via access(2) with the `F_OK` mode. Otherwise, this function converts the permission mask to an access vector using the `file_mask_to_av` function, and calls `inode_has_perm` with the appropriate parameters. Table 19 specifies the SELinux permission that is checked for each permission mask flag when checking access to a directory. Table 20 provides the corresponding permission information when checking access to a non-directory file.

In Table 19, notice that a write permission mask causes the general `write` permission to be checked. This hook function cannot distinguish among the various kinds of modification operations on directories, so it cannot use the finer-grained permissions (`add_name`, `remove_name`, or `reparent`). Hence, directory modifications require both the general `write` permission and the appropriate finer-grained permission to be granted between the task and the inode. The general `write` permission check could be omitted from this hook, but it is performed to ensure that all directory modifications are mediated by the policy.

Table 19. Directory Permission Checks

Mask	Permission
`MAY_EXEC`	search
`MAY_READ`	read
`MAY_WRITE`	write

In Table 20, notice that a separate `MAY_APPEND` permission mask and `append` permission are listed. This permission mask was added by the LSM kernel patch and is used (along with MAY_WRITE) when a file is opened with the `O_APPEND` flag. This allows the security module to distinguish append access from general write access. The `selinux_file_fcntl` hook ensures that the `O_APPEND` flag is not subsequently cleared unless the process has `write` permission to the file.

Table 20. Non-Directory Permission Checks

Mask	Permission
`MAY_EXEC`	execute
`MAY_READ`	read
`MAY_APPEND`	append
`MAY_WRITE`	write

14.2.7. selinux_inode_setxattr

This hook function is called to check permissions prior to setting an extended attribute (xattr) for an inode. If the attribute is not the SELinux attribute but is in the security namespace, then the function checks CAP_SYS_ADMIN to protect the security namespace for unprivileged processes. If the attribute is not in the security namespace at all, then this function simply checks the `setattr` permission to the inode.

If the attribute is the SELinux attribute, then this function first checks whether mountpoint labeling is being used, in which case it immediately returns an error indicating that setxattr is not supported. Otherwise, the function checks whether the process owns the file and if not, checks CAP_FOWNER capability, in order to provide a DAC restriction over file relabeling. The function then applies a series of mandatory permission checks for file relabeling, as summarized in Table 21. It also invokes the security server's `security_validate_transition` function to apply any checks based on all three security contexts (the old file context, the new file context, and the process context) together. This function was introduced as part of the enhanced MLS support to support MLS upgrade and downgrade checks, but can be generally applied for other kinds of policy logic as well.

Table 21. Setxattr Permission Checks

Source	Target	Permission(s)
Current	OldFileContext	relabelfrom
Current	NewFileContext	relabelto
File	Filesystem	associate

14.2.8. Other inode access control hook functions

The remaining inode hook functions are called to check permissions for various operations. Since each of these remaining hook functions only require a single permission between the current task and the file, the permission checks are all described in Table 22.

Table 22. Remaining Inode Hook Permission Checks

Hook	Permission
selinux_inode_readlink	read
selinux_inode_follow_link	read
selinux_inode_setattr	setattr or write
selinux_inode_getattr	getattr
selinux_inode_getxattr	getattr
selinux_inode_listxattr	getattr

Of these hooks, only two require further description. First, the `selinux_inode_setattr` hook checks the `setattr` permission to the file if setting the file mode, uid, gid or explicitly setting the timestamps to a particular value via utimes; otherwise, it merely checks write permission. Separate permissions could be defined for different kinds of setattr operations, e.g. chown, chmod, utimes, truncate. However, this level of distinction does not seem to be necessary to support mandatory access control policies. Second, the `selinux_inode_follow_link` hook checks the same permission as the `selinux_inode_readlink` hook, i.e. read permission. While this is correct from an information flow perspective and while even reading a malicious symlink may constitute a hazard (e.g. for realpath(3)), it may be desirable in the future to introduce a separate follow permission to allow a trusted process to see all symlinks (e.g. for ls -l) without necessarily being able to follow them (in order to protect against malicious symlinks).

15. File Hook Functions

The SELinux file hook functions manage the security fields of file structures and perform access control for file operations. Each file structure contains state such as the file offset and file flags for an open file. Since file descriptors may be inherited across `execve` calls and may be transferred through IPC, they can potentially be shared among processes with different security attributes, so it is desirable to separately label these structures and control the use of them. Additionally, it is necessary to save task security information in these structures for `SIGIO` signals.

15.1. Managing File Security Fields

15.1.1. File Security Structure

The file_security_struct structure contains security information for file objects. This structure is defined as follows:

```
struct file_security_struct {
        struct file *file;
        u32 sid;
        u32 fown_sid;
};
```

Table 23. file_security_struct

Field	Description
file	Back pointer to the associated file.
sid	SID of the open file descriptor.
fown_sid	SID of the file owner; used for `SIGIO` events.

15.1.2. file_alloc_security and file_free_security

The `file_alloc_security` and `file_free_security` helper functions are the primitive allocation functions for file security structures. In addition to the general security field management, `file_alloc_security` associates the open file with the SID of the allocating task. The `selinux_file_alloc_security` and `selinux_file_free_security` hook functions simply call the helper functions.

15.1.3. selinux_file_set_fowner

This hook function is called to save security information about the current task in the file security structure for later use by the `selinux_file_send_sigiotask` hook. One example of where this hook is called is the `fcntl` call for the `F_SETOWN` command. This hook saves the SID of the current task in the *`fown_sid`* field of the file security structure.

15.2. Controlling File Operations

15.2.1. file_has_perm

This helper function checks whether a task can use an open file descriptor to access a file in a given way. It takes the task, the file, and the requested file permissions as parameters. This function first sets up the auxiliary audit data. It then calls the AVC to check `use` permission between the task and the file descriptor. If this permission is granted, then this function also checks the requested permissions to the file using the `inode_has_perm` helper function. In some cases (e.g. certain ioctl and fcntl commands), this helper function is called with no requested file permissions in order to simply check the ability to use the descriptor. In these cases, the latter check is omitted.

15.2.2. selinux_file_permission

This hook function is called by operations such as `read`, `write`, and `sendfile` to revalidate permissions on use to support privilege bracketing or policy changes. It takes the file and permission mask as parameters. If the permission mask is null (an existence test), then the function returns success immediately. Otherwise, if the `O_APPEND` flag is set in the file flags, then this hook function first sets the `MAY_APPEND` flag in permission mask. This function then converts the permission mask to an access vector using the `file_mask_to_av` function, and calls `file_has_perm` with the appropriate parameters.

15.2.3. selinux_file_ioctl

This hook function is called by the `ioctl` system call. It calls `file_has_perm` with a requested file permission based on the command argument. For some commands, no file permission is specified so only the `use` permission is checked. The generic `ioctl` file permission is used for commands that are not specifically handled. Table 24 shows the permission checks performed for each command.

Table 24. I/O Control Permission Checks

Command	Source	Target	Permission(s)
FIONREAD FIBMAP FIGETBSZ EXT2_IOC_GETFLAGS EXT2_IOC_GETVERSION	Current	FileDescriptor File	use getattr

Command	Source	Target	Permission(s)
EXT2_IOC_SETFLAGS EXT2_IOC_SETVERSION	Current	FileDescriptor File	use setattr
FIONBIO FIOASYNC	Current	FileDescriptor	use
Other	Current	FileDescriptor File	use ioctl

15.2.4. file_map_prot_check

This helper function is called by the `selinux_file_mmap` and the `selinux_file_mprotect` hook functions to apply permission checks for attempts to create or change the protection of memory mappings. The function first checks whether the caller is attempting to make executable an anonymous mapping or a private file mapping that will also be writable. If so, it applies the process `execmem` permission check to control the ability to execute arbitrary code from memory.

If a file is being mapped, then this function calls the `file_has_perm` with a set of permissions based on a flag indicating whether the mapping is shared and the requested protection. Since read access is always possible with a mapping, the `read` permission is always required. The `write` permission is only checked if the mapping is shared and `PROT_WRITE` was requested. The `execute` permission is only checked if `PROT_EXEC` was requested.

It should be noted that the protection on a mapping may subsequently become invalid due to a file relabel or a change in the security policy. Hence, support for efficiently locating and invalidating the appropriate mappings upon such changes is needed to support full revocation. This support has not yet been implemented for the SELinux security module.

15.2.5. selinux_file_mmap

This hook function is called to check permission when creating a mapping. The hook function determines whether the mapping will be shared based on the provided flags and calls the `file_map_prot_check` helper.

15.2.6. selinux_file_mprotect

This hook function is called to check the requested new protection for an existing mapping. If the caller is attempting to make the mapping executable, this function first applies several specialized checks. If the mapping is in the brk region, then the process `execheap` permission is checked to control attempts to make the heap executable, which should normally never occur and is not portable; such memory if needed should be explicitly allocated via mmap. If the mapping is a private file mapping that has had some copy-on-write done, indicating that it may include modified content, then this function performs a file `execmod` permission check. Typically, this should only occur for text relocations, which if possible should be eliminated from the program or DSO. If the mapping is in the main process stack, this function checks the process `execstack` permission to control attempts to make the stack executable; as with execheap, such memory if needed should be explicitly allocated via mmap. This function then determines whether the mapping is shared based on the flags in the vm_area_struct and calls the `file_map_prot_check` helper to complete checking. Note that in the execstack case, this will also trigger an `execmem` check, so both permissions would have to be allowed in order to permit making the stack executable; however, in practice, the more likely situation is that one would allow execmem to a particular program to permit legitimate runtime code generation while denying execstack to prevent making its stack executable.

15.2.7. selinux_file_lock

This hook function is called to check permissions before performing file locking operations. It calls `file_has_perm` with the `lock` permission.

15.2.8. selinux_file_fcntl

This hook function is called by the `fcntl` system call. It calls `file_has_perm` with a requested file permission based on the command parameter. The basic permission checks performed for each command are shown in Table 25.

Table 25. File Control Permission Checks

Command	Source	Target	Permission(s)
F_SETFL F_SETOWN F_SETSIG F_GETFL F_GETOWN F_GETSIG	Current	FileDescriptor	use
F_GETLK F_SETLK F_SETLKW F_GETLK64 F_SETLK64 F_SETLKW64	Current	FileDescriptor File	use lock

In addition to these basic checks, the `write` permission is checked if the `F_SETFL` command is used to clear the `O_APPEND` flag. This ensures that a process that only has `append` permission to the file cannot subsequently obtain full write access after opening the file.

15.2.9. selinux_file_send_sigiotask

This hook function is called to check whether a signal generated by an event on a file descriptor can be sent to a task. This function is sometimes called from interrupt. It is passed the target task, a file owner structure and the signal that would be delivered (or 0 if SIGIO is to be used as the default). Since the file owner structure is embedded in a file structure, the file structure and its security field can be extracted by the hook function. The hook function calls the AVC to check the appropriate signal permission between the `fown_sid` in the file security structure and the target task SID.

15.2.10. selinux_file_receive

This hook function is called to check whether the current task can receive an open file descriptor that was sent via socket IPC. This function calls the `file_to_av` function to convert the file flags and mode to an access vector and then calls `file_has_perm` to check that the receiving task has these permissions to the file. If this hook returns an error, then the kernel will cease processing the message and will pass a truncated message to the receiving task.

15.2.11. selinux_quota_on

This hook function is called to check permissions when quotas are enabled to a particular quota file. It calls `file_has_perm` to check `quotaon` permission to the file.

16. System V IPC Hook Functions

The SELinux System V Inter-Process Communication (IPC) hook functions manage the security fields and perform access control for System V semaphores, shared memory segments, and message queues. This section describes these hooks and their helper functions.

16.1. Managing System V IPC Security Fields

16.1.1. IPC Security Structure

The ipc_security_struct structure contains security information for IPC objects. This structure is defined as follows:

```
struct ipc_security_struct {
        struct kern_ipc_perm *ipc_perm;
        security_class_t sclass;
        u32 sid;
};
```

Table 26. ipc_security_struct

Field	Description
ipc_perm	Back pointer to the associated kern_ipc_perm.
sclass	Security class for the IPC object (see Section 16.1.2).
sid	SID for the IPC object.

Likewise, the msg_security_struct structure contains security information for IPC message objects. This structure is defined as follows:

```
struct msg_security_struct {
        struct msg_msg *msg;
        u32 sid;
};
```

Table 27. msg_security_struct

Field	Description
msg	Back pointer to the associated IPC message;
sid	SID for the IPC message.

16.1.2. ipc_alloc_security and ipc_free_security

The `ipc_alloc_security` and `ipc_free_security` helper functions are the primitive allocation functions for the security structures for semaphores, shared memory segments, and message queues. The kernel data structures for these objects share a common substructure, kern_ipc_perm, and the security field is located in this shared substructure; a single set of helper functions can be used for all three object types. A new IPC object inherits its SID from the creating task. The security class for the IPC object is passed by the caller; it will be one of `SECCLASS_MSGQ`, `SECCLASS_SEM`, or `SECCLASS_SHM`.

The `ipc_alloc_security` helper function is called by the following allocation hook functions:

- `selinux_sem_alloc_security`
- `selinux_shm_alloc_security`
- `selinux_msg_queue_alloc_security`

After calling the helper, the allocation hook functions set up auxiliary audit data and then call the AVC to check the `create` permission between the current task and the IPC object. Hence, these hook functions have the unusual property of being used both for allocation and a permission check. Using two separate hooks for this purpose would be cleaner but inefficient, since they would both be called at the same point.

The `ipc_free_security` function is called upon a permission denial by the allocation hook functions as well as by the following deallocation hook functions:

- `selinux_sem_free_security`
- `selinux_shm_free_security`
- `selinux_msg_queue_free_security`

These deallocation hook functions do not perform any other processing.

16.1.3. msg_msg_alloc_security and msg_msg_free_security

The `msg_msg_alloc_security` and `msg_msg_free_security` helper functions are the primitive allocation functions for the security structures for individual messages on a message queue. These helper functions provide all of the processing for the selinux_msg_msg_alloc_security and selinux_msg_msg_free_security hook functions. These helper functions simply provide the standard processing for primitive allocation functions, and initialize the message SID to the unlabeled SID.

16.2. Controlling General IPC Operations

This section describes the helper and hook functions for controlling general IPC operations. Although the allocation functions do perform a create permission check, they are not listed here since they were discussed in the previous section.

16.2.1. ipc_has_perm

This helper function sets up the auxiliary audit data information and calls the AVC to check whether the current task has a particular permission to an IPC object. The explicit passing of the security class of the IPC object is a legacy of the earlier handling for pre-existing objects prior to SELinux initialization via precondition functions and could be removed, using the sclass field from the security structure instead.

16.2.2. selinux_ipc_permission

This hook function is called from the kernel ipcperms function, so it is called prior to all IPC operations that will read or modify the IPC object. This hook function checks unix_read and/or unix_write permission to the IPC object based on the flag, as shown in Table 28. These permissions provide a coarse-grained equivalent to the Unix permissions, whereas the other IPC hooks check finer-grained permissions. These coarse-grained permission checks are not strictly necessary, but ensure that all IPC accesses are mediated by the policy.

Table 28. ipc_permission Permission Checks

Flag	Permission
S_IRUGO	unix_read
S_IWUGO	unix_write

16.2.3. selinux_*_associate

When a task attempts to obtain an IPC object identifier for an existing object via one of the *get calls, the kernel calls the corresponding associate hook function for the object type. The SELinux IPC associate hook functions are:

- selinux_sem_associate
- selinux_shm_associate

- `selinux_msg_queue_associate`

These hook functions check `associate` permission between the current task and the IPC object.

16.3. Controlling Semaphore Operations

16.3.1. selinux_semctl

This hook function checks permissions before performing an operation on the specified semaphore; the specific permission is determined by the operation being performed. The permissions required for each operation are shown in Table 29.

Table 29. Semaphore Control Permissions

Operation	Source	Target	Permission
IPC_INFO SEM_INFO	Current	System	ipc_info
GETPID GETNCNT GETZCNT	Current	Sem	getattr
GETVAL GETALL	Current	Sem	read
SETVAL SETALL	Current	Sem	write
IPC_RMID	Current	Sem	destroy
IPC_SET	Current	Sem	setattr
IPC_STAT SEM_STAT	Current	Sem	getattr, associate

16.3.2. selinux_semop

This hook function checks permissions for semaphore operations. It always checks `read` permission between the current task and the semaphore. If the semaphore value is being altered, it also checks `write` permission between the current task and the semaphore. Notice that these permissions are different from the `unix_read` and `unix_write` permissions checked by `selinux_ipc_permission`.

16.4. Controlling Shared Memory Operations

16.4.1. selinux_shm_shmctl

This hook function checks permissions before performing an operation on the specified shared memory region; the specific permission is determined by the operation being performed. The permissions required for each operation are shown in Table 30.

Table 30. Shared Memory Control Permissions

Operation	Source	Target	Permission
IPC_INFO SHM_INFO	Current	System	ipc_info
IPC_STAT SHM_STAT	Current	Shm	getattr, associate
IPC_SET	Current	Shm	setattr
SHM_LOCK SHM_UNLOCK	Current	Shm	lock
IPC_RMID	Current	Shm	destroy

16.4.2. selinux_shm_shmat

This hook function checks permissions for shared memory attach operations. It always check `read` permission between the current task and the shared memory object. If the `SHM_RDONLY` flag was not specified, then it also checks `write` permission between the current task and the shared memory object. Notice that these permissions are different from the `unix_read` and `unix_write` permissions checked by `selinux_ipc_permission`.

16.5. Controlling Message Queue Operations

16.5.1. selinux_msg_queue_msgctl

This hook function checks permissions before performing an operation on the specified message queue; the specific permission is determined by the operation being performed. The permissions required for each operation are shown in Table 31.

Table 31. Message Queue Control Permissions

Operation	Source	Target	Permission
IPC_INFO MSG_INFO	Current	System	ipc_info

Operation	Source	Target	Permission
IPC_STAT MSG_STAT	Current	MessageQueue	getattr, associate
IPC_SET	Current	MessageQueue	setattr
IPC_RMID	Current	MessageQueue	destroy

16.5.2. selinux_msg_queue_msgsnd

This hook function is called by the `msgsnd` system call to check the ability to place an individual message on a message queue. It performs three permission checks, involving the current task, the message queue, and the individual message. These checks are shown in Table 32. This hook function also sets the SID on the message if it is unlabeled. It calls the `security_transition_sid` interface of the security server to obtain a SID based on the SID of the task and the SID of the message queue.

Table 32. Message Send Permissions

Source	Target	Permission
Current	MessageQueue	write
Current	Message	send
Message	MessageQueue	enqueue

16.5.3. selinux_msg_queue_msgrcv

This hook function can be called by either the `msgsnd` system call (for a pipelined send) or by the `msgrcv` system call to check the ability to receive an individual message from a message queue. Hence, the receiving task may not be the current task and is explicitly passed to the hook. This hook function performs two permission checks, involving the receiving task, the message queue, and the individual message. These permission checks are shown in Table 33. It is important to note that an error return from this hook simply causes the individual message to be ignored in the same manner as if it had the wrong message type. Hence, access denials on individual messages are not propagated to the calling process and may cause the calling process to block waiting for messages that are accessible.

Table 33. Message Receive Permissions

Source	Target	Permission
ReceiverTask	MessageQueue	read
ReceiverTask	Message	receive

17. Socket Hook Functions

The SELinux socket hook function implementations manage the security fields of socket objects and perform access control for socket operations. This section describes these hooks and their helper functions.

17.1. Managing Socket Security Fields

17.1.1. Socket Security Structure

Each user space socket structure (struct socket) has an associated inode structure, so the inode security structure is extensively used for socket objects as well. See Section 14.1 for a discussion of inode security structure and associated functions. A security field also exists in the network layer socket structure (struct sock), but this field can only be safely used for local/Unix domain sockets presently. A change to the TCP code would be required to ensure proper handling of this field for newly created server sockets created by a connection; such a change was included in the LSM kernel patch, but did not make it into the mainline kernel due to the rejection of the LSM networking hooks.

For local/Unix domain sockets, the sk_security_struct is used to store security information about the peer during connection establishment when the user socket is not yet allocated for the new connection. This structure is defined as follows:

```
struct sk_security_struct {
        struct sock *sk;
        security_id_t peer_sid;
}
```

Table 34. sk_security_struct

Field	Description
sk	Back pointer to the associated sock structure.
peer_sid	SID of the peer socket.

17.1.2. sk_alloc_security and sk_free_security

The `sk_alloc_security` and `sk_free_security` helper functions are the primitive allocation functions for sock security structures. They immediately return if the socket family is anything other than the local/Unix domain, as they cannot safely handle other kinds of sockets. Otherwise, they perform the usual allocation and initialization of the security structure.

17.1.3. selinux_socket_getpeersec

This hook function is called to handle the SO_PEERSEC getsockopt option. It first checks whether the socket is local/Unix domain, and if not, returns an error. Otherwise, it extracts the peer SID from the sock security structure, converts it to a context, and copies it to the user buffer.

17.1.4. selinux_socket_post_create

After a socket structure has been successfully created, this hook function is called to setup the inode security structure for the socket . It set the security class using `socket_type_to_security_class`, as shown in Table 35. The netlink socket class is further partitioned based the netlink protocol to support fine-grained control. If the socket does not match any of the specified types, it defaults to the generic socket security class. The hook function then sets the inode SID. The hook function is passed a flag indicating whether the socket is being created for kernel-internal use (e.g. for RPC) or for userspace. If the socket is for kernel-internal use, then it is labeled with the kernel initial SID. Otherwise, it is labeled with the SID of the creating task.

Table 35. Socket Security Classes

Protocol Family	Type	Protocol	Security Class
`PF_UNIX`	SOCK_STREAM SOCK_SEQPACKET	ignored	unix_stream_socket
`PF_UNIX`	`SOCK_DGRAM`	ignored	unix_dgram_socket
`PF_INET/PF_INET6`	`SOCK_STREAM`	IPPROTO_IP IPPROTO_TCP	tcp_socket
`PF_INET/PF_INET6`	`SOCK_DGRAM`	IPPROTO_IP IPPROTO_UDP	udp_socket
`PF_INET/PF_INET6`	any other value	any other value	rawip_socket
`PF_NETLINK`	ignored	ignored	netlink_*_socket
`PF_PACKET`	ignored	ignored	packet_socket
`PF_KEY`	ignored	ignored	key_socket

17.1.5. selinux_socket_accept

This hook function is called after a new socket has been created for the connection but prior to calling the protocol family's accept function. In addition to checking permission (discussed further in Section 17.2), this hook function sets the SID and security class in the inode security structure for the new socket. The new socket inherits the SID and security class of the listening socket. The new socket initialization must occur in this hook, since traffic can occur on the socket before the `post_accept` hook is called.

17.2. Controlling Socket Operations

17.2.1. socket_has_perm

This helper function checks whether a task has a particular permission to a socket. It first checks whether the socket is for kernel-internal use, and if so, returns success immediately. Otherwise, it sets up the auxiliary audit data and calls the AVC to check the permission.

17.2.2. General Socket Layer Hooks

The socket layer access control hook functions first check a permission between the current task and the socket using the `socket_has_perm` helper function. Some of the hook functions perform additional processing. The hook functions and the initial permission that they check are shown in Table 36. Any additional processing for the hook functions is then described after this table.

Table 36. Socket Layer Hook Permission Checks

Hook Function	Source	Target	Permission
selinux_socket_create	Current	NewSocket	create
selinux_socket_bind	Current	Socket	bind
selinux_socket_connect	Current	Socket	connect
selinux_socket_listen	Current	Socket	listen
selinux_socket_accept	Current	Socket	accept
selinux_socket_sendmsg	Current	Socket	write
selinux_socket_recvmsg	Current	Socket	read
selinux_socket_getsockname	Current	Socket	getattr
selinux_socket_getpeername	Current	Socket	getattr
selinux_socket_setsockopt	Current	Socket	setopt
selinux_socket_getsockopt	Current	Socket	getopt
selinux_socket_shutdown	Current	Socket	shutdown

The `selinux_socket_bind` hook function performs an additional `name_bind` permission check between the socket and the SID associated with the port number for ports that are outside the range used to automatically bind. It also performs an additional `node_bind` permission check between the socket and the SID associated with the IP address.

The `selinux_socket_connect` hook function performs an additional `name_connect` permission check between the socket and the SID associated with the port number for TCP sockets. This check provides control over outbound TCP connections to particular ports distinct from the general controls over sending and receiving packets.

17.2.3. Controlling Receipt of Packets

The `selinux_socket_sock_rcv_skb` hook function is called by the `sk_filter` kernel function prior to applying any socket filters to control receipt of individual packets on a socket at a point where the destination socket and the receiving network device information is available. The hook function begins by checking whether the socket family corresponds with IPv4 or IPv6 and returning success immediately otherwise. It then checks for mapped IPv4 packets arriving via IPv6 sockets and adjusts the family information accordingly for later use in translation of the headers.

Unlike the previously discussed socket hook functions, this hook is passed a pointer to a network layer socket (sock) structure rather than a userspace socket structure. This hook function must (while holding the appropriate lock) first dereference the *`socket`* field of the sock structure and then dereference the `inode` field of the resulting socket structure in order to obtain security information about the receiving socket. However, security information is not always available, e.g. the socket may not be presently associated with an userspace socket (e.g. new server socket that has not yet been accepted, or a userspace socket that has been closed).

After obtaining the socket security information, the hook function must also obtain security information for the receiving network device. It calls the `sel_netif_sids` function to obtain the interface SID associated with the device. It then determines the right set of permissions to check based on the socket class, sets up auxiliary audit data, and calls `selinux_parse_skb` to parse the headers for address information to include in the audit data. It then performs permission checks between the socket SID and the SIDs associated with the receiving network interface, the remote host, and the source port, as shown in Table 37. Note that these permission checks differ from the original set of permission checks for packet receipt prior to the redesign for Linux 2.6.

Table 37. Permission Checks for Receiving a Packet on a Socket

Source	Target	Permission(s)
Socket	NetworkInterface	udp_recv tcp_recv rawip_recv
Socket	RemoteNode (Host)	udp_recv tcp_recv rawip_recv
Socket	SourcePort	recv_msg

17.2.4. Hooks for Unix Domain Socket IPC

LSM places calls to two hooks, `unix_stream_connect` and `unix_may_send`, within the Unix domain socket code to provide consistent control over Unix domain socket IPC. These hooks are placed into the Unix domain socket code in order to have access to the destination socket, which is not available to the socket layer hooks. For sockets that use the file namespace, the inode hook functions could be used to control IPC, but this would not address sockets that use the abstract namespace. Hence, these two hooks were added by LSM.

The `selinux_socket_unix_stream_connect` hook function is called for Unix stream connections. It checks the `connectto` permission between the client socket and the listening socket. It also sets the peer SID fields in each of the peer sockets' security structures for later use by `selinux_socket_getpeersec`. The `selinux_socket_unix_may_send` hook function is called for Unix datagram communications. It checks the `sendto` permission between the sending socket and the receiving socket. These permission checks are summarized in Table 38.

Table 38. Unix Domain Permission Checks

Hook	Source	Target	Permission
unix_stream_connect	ClientSocket	ServerSocket	connectto

Hook	Source	Target	Permission
unix_may_send	SendingSocket	ReceivingSocket	sendto

18. IP Networking Hook Functions

The LSM kernel patch added a set of security fields and hooks to allow management of security data for several network-related data structures, including network buffers, network devices, and network layer sockets. It also added a number of hooks to the IP network stack to support IP packet lifecycle management, particularly to support packet labeling using CIPSO-style options, that could not be directly supported via the existing NetFilter hooks. The LSM-based SELinux network access control functionality was originally implemented using these security fields and hooks as well as several NetFilter hooks. However, the LSM security fields and hooks for networking were not accepted for inclusion in Linux 2.6. As a result, the SELinux network access controls were redesigned and implemented using only the socket layer hooks and NetFilter hooks, and some functionality such as packet labeling was dropped from SELinux. This section describes the SELinux NetFilter hook functions.

* *Note: The preceding statements are historical and no longer apply to modern SELinux systems, which do include a set of network hooks and support packet labeling.*

The SELinux IPv4 and IPv6 NetFilter hook functions, `selinux_ipv4_postroute_last` and `selinux_ipv6_postroute_last`, perform permission checks for outgoing packets after routing has occurred. Incoming packets are mediated by the `selinux_socket_sock_rcv_skb` LSM hook, which is described in Section 17.2.3. Both of the NetFilter hook functions call a common helper, `selinux_ip_postroute_last`, to perform all processing.

The helper function begins by extracting the socket security information from the associated inode security structure. After obtaining the socket security information, the hook function must also obtain security information for the sending network device. It calls the `sel_netif_sids` function to obtain the interface SID associated with the device. It then determines the right set of permissions to check based on the socket class, sets up auxiliary audit data, and calls `selinux_parse_skb` to parse the headers for address information to include in the audit data. It then performs permission checks between the socket SID and the SIDs associated with the sending network interface, the remote host, and the destination port, as shown in Table 39. Note that these permission checks differ from the original set of permission checks for packet receipt prior to the redesign for Linux 2.6.

Table 39. Permission Checks for Sending a Packet from a Socket

Source	Target	Permission(s)
Socket	NetworkInterface	udp_send tcp_send rawip_send

Source	Target	Permission(s)
Socket	RemoteNode (Host)	udp_send tcp_send rawip_send
Socket	DestinationPort	send_msg

19. Miscellaneous Hook Functions

This section describes miscellaneous hook functions that do not fit into one of the prior sections.

19.1. Capability-Related Hook Functions

19.1.1. selinux_capable

This hook function is called by the kernel to determine whether a particular Linux capability is granted to a task. After calling the secondary security module to perform the ordinary Linux capability test or superuser test, this hook function calls the `task_has_capability` helper function to check the corresponding SELinux capability permission. Hence, the Linux capability must be granted by both the secondary security module and by SELinux.

19.1.2. selinux_capget

This hook function is called by the kernel to get the capability sets associated with a task. It first checks `capget` permission between the current and target tasks. If this permission is granted, it then calls the secondary security module to obtain the capability sets, since SELinux does not maintain this information. Note that the returned capability sets are not modified to remove capabilities that would be denied by SELinux.

19.1.3. selinux_capset_check

This hook function is called by the kernel to check permission before setting the capability sets associated with a task. It calls the secondary module to allow it to apply the normal Linux capability checking logic, and then checks `capset` permission between the current and target task. SELinux does not perform any checks on the individual capabilities being set, since it revalidates each capability on use in the `selinux_capable` hook.

19.1.4. selinux_capset_set

This hook function is called by the kernel to set the capability sets associated with a task. It simply calls the secondary module to set the capability sets, since SELinux does not maintain this information.

19.1.5. selinux_netlink_send

This hook function is called to perform permission checking and to set the effective capability set in the control buffer for a netlink message when the message is sent. The function first calls the secondary module to initialize the effective capability set based on the sending task. It then calls the AVC to compute the set of capabilities that would be allowed by SELinux and intersects this set with the effective capability set in the control buffer. Finally, if the policy supports the fine-grained netlink classes and permissions, this hook function calls `selinux_nlmsg_perm` to apply further permission checks based on a mapping of netlink message types to read and write information flows (i.e. observing information or modifying information).

19.1.6. selinux_netlink_recv

This hook function is called to check permission when a netlink message is received that requires privilege. It checks the effective capability set associated with the netlink message to see if `CAP_NET_ADMIN` is set.

19.1.7. selinux_vm_enough_memory

This hook function is called to check permissions and perform accounting when allocating a mapping. It was initially made into a security hook to avoid generating spurious audit messages upon checking CAP_SYS_ADMIN to determine whether to reserve some memory. The hook function calls the secondary module's capable function to check whether the task has CAP_SYS_ADMIN, and if so, it calls the AVC to check whether SELinux would allow this capability as well, using an interface that avoid audit generation. The function then calls the kernel `__vm_enough_memory` function with a flag indicating whether the capability was granted.

19.2. Sysctl Hook Function

The `selinux_sysctl` hook function checks permission for the current task to access a sysctl entry. It calls the `selinux_proc_get_sid` helper function to obtain the SID associated with the sysctl entry based on the proc_dir_entry tree. This is also used by `inode_doinit_with_dentry` for other procfs inodes, as discussed in Section 14.1.3. If no match is found, then the hook function defaults to the sysctl initial SID.

The hook function then performs a permission check based on the requested operation, treating the sysctl entry as a directory for search operations and as a file for read or write operations on a variable. Table 40 shows the permission checks associated with each requested operation.

Table 40. sysctl Permission Checks

Operation Value	Source	Target	Permission
`1`	Current	Entry	Search
`4`	Current	Entry	read
`2`	Current	Entry	write

19.3. Syslog Hook Function

The `selinux_syslog` hook function checks that the current task has permission to perform a given system logging command. For operations `3` (read last kernel messages) and `10` (return size of log buffer), the `syslog_read` system permission is checked. For operations that control logging to the console, the `syslog_console` system permission is checked. All other operations (including unknown ones) are checked with `syslog_mod` system permission.

References

[LoscoccoFreenix2001] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, The USENIX Association, June 2001.

[LoscoccoNSATR2001] Peter Loscocco and Stephen Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", *NSA Technical Report*, February 2001.

[LoscoccoNISS1998] Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, S. Turner, and John Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments", *Proceedings of the 21st National Information Systems Security Conference*, October 1998.

[SpencerUsenixSec1999] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies", *Proceedings of the Eighth USENIX Security Symposium*, The USENIX Association, August 1999.

[FIPS188] *FIPS PUB 188: Standard Security Label for Information Transfer*, U.S. Dept. of Commerce / National Institute of Standards and Technology, September 6, 1994.

[MorrisSeloptOverview2002] James Morris, "Overview of SELinux Labeled Networking Support via CIPSO/FIPS-188 IP Options", *selopt-overview.txt*, February 2002.

[MorrisSCMP2001] James Morris, "Security Context Mapping Protocol", *scmp-draft.txt*, December 30, 2001.