Implementing SELinux as a Linux Security Module

Stephen Smalley

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: May 2002
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 WireX to develop such a framework. LSM is 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 patch is currently tracking the 2.4 series and is targeted for integration into the 2.5 development series. The LSM kernel patch is available from the LSM web site (http://lsm.immunix.org). A brief overview of the LSM framework is available in the `Documentation/DocBook/lsm.tmpl` file in the LSM-patched kernel tree, and detailed documentation for each LSM hook is available in the `include/linux/security.h` file in the same tree.

The SELinux implementation has been reworked by NAI Labs to use the LSM patch 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 and exit code, its support for stacking with other security modules, and its approach for implementing the new SELinux system calls. 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.

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) kernel patch. This section contains an edited excerpt from the `Documentation/DocBook/lsm.tmpl` file in the LSM-patched kernel tree.

The LSM kernel patch provides a general kernel framework to support security modules. In particular, the LSM framework is primarily focused on supporting access control modules, although future development is likely to address other security needs such as auditing. By itself, the framework does not provide any additional security; it merely provides the infrastructure to support security modules. The LSM kernel patch 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 kernel patch 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, and adds a general `security` system call to support new system calls for security-aware applications.

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. For packet and network device security information, security fields were added to struct sk_buff and struct net_device. For System V IPC security information, security fields were added to struct kern_ipc_perm and struct msg_msg; additionally, the definitions for struct msg_msg, struct msg_queue, and struct shmid_kernel were moved to header files (`include/linux/msg.h` and `include/linux/shm.h` as appropriate) to allow the security modules to use these definitions.

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. At present, this structure consists of a collection of substructures that group related hooks based on the kernel object (e.g. task, inode, file, sk_buff, etc) as well as some top-level hook function pointers for system operations. This structure is likely to be flattened in the future for performance. The hook calls can be easily found in the kernel code by looking for the string "security_ops->".

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 into substructures 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 `post_lookup` hook in struct inode_security_ops. This hook is used to set security information for inodes after successful lookup operations. An example of the second category of hooks is the `permission` hook in struct inode_security_ops. This hook checks permission when accessing an inode.

LSM adds a general `security` system call that simply invokes the `sys_security` hook. This system call and hook permits security modules to implement new system calls for security-aware applications. The interface is similar to socketcall, but also has an *`id`* to help identify the security module whose call is being invoked.

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 a local, non-persistent integer that is mapped by the security server to a security context. 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 maintains a persistent label mapping in each filesystem that maps inodes to integer persistent security identifiers (PSIDs) and maps PSIDs to security contexts.

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.

In addition to returning a decision for the permission check, the AVC returns a reference to the entry in the cache that contained the decision. The policy enforcement code can save this reference with the object and provide it as a hint on subsequent permission checks to optimize the lookup. These references are referred to as AVC entry references. The references are revalidated on use, so if the SID of the subject or object has changed or if the referenced entry has been invalidated due to a policy change, the AVC will look up the correct entry or obtain a new one from the security server and return an updated reference.

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, handling pre-existing subjects and objects, stacking with the capabilities module, reimplementing the extended system calls, 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. At present, the SELinux security module does directly store a single word (a single SID) in the security field of one of the kernel data structures, the struct linux_binprm structure, but this may be changed in the future. This is discussed further in Section 12.1. The SELinux security module uses a dynamically-allocated security structure for the security fields of the other kernel data structures.

5.1.3. Handling Pre-Existing Subjects and Objects

With the original SELinux kernel patch, it was possible to ensure that all subjects and objects are labeled when they are initialized or created. The LSM-based SELinux security module must handle subjects and objects in the system that were created prior to module initialization. Some tasks and objects (e.g. the procfs root inode) are created prior to module initialization even when the module is compiled into the

kernel, so there are always some pre-existing subjects and objects that must be handled. When the module is dynamically loaded into a kernel, the situation is even more complicated.

The LSM-based SELinux security module addresses this problem by defining a precondition function for each kernel object that dynamically handles the allocation and initialization of the corresponding security structure if it is not already set, and by calling this precondition function prior to any attempts to dereference the security field. These functions are described in general in Section 10.2 and in more detail in the individual hook subsections. However, these functions can not always retroactively determine the correct security information for a pre-existing subject or object, so it is recommended that the SELinux security module always be built into the kernel.

5.1.4. 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.

5.1.5. Reimplementing the Extended System Calls

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. In the LSM-based SELinux security module, these extended system calls were implemented by passing SID parameters to and from the hook functions via fields in the current task's security structure. This is discussed further in Section 9.

5.1.6. 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, changing the file descriptor inheritance controls, and changing the controls over process tracing and state sharing when a new program is executed. 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 12.1.2.

5.2.2. File Descriptor Inheritance

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.

5.2.3. Process Tracing and State Sharing

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 12.1.3.

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 extending the persistent label mapping to be filesystem-independent, reimplementing file labeling support for pseudo filesystem types, leveraging the hook in the existing `permission` function, revising the file descriptor permission checks, changing the `open_secure` interface, and eliminating the pipe security class. Each change is described below.

5.3.1. Persistent Labeling

In the original SELinux kernel patch, the persistent label mapping in each filesystem stored a mapping from 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 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 should allow SELinux to support other file system types more easily, but has disadvantages in terms of performance and consistency. Naturally, if support for extended attributes becomes integrated into the mainstream Linux kernel, SELinux will be modified to take advantage of it when extended attributes are supported by the filesystem.

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 does not provide filesystem-specific hooks, the LSM-based SELinux security module had to reimplement this functionality using the hooks in the VFS layer. In addition to reimplementing the labeling functionality for these filesystem types, labeling support for the tmpfs and devfs filesystems was also added to the LSM-based SELinux security module. The handling for these pseudo filesystem types is described in Section 14.1.3.

5.3.3. Leveraging `permission`

As discussed in Section 5.1.6, 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 that was incorporated into the LSM kernel patch.

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. `open_secure` Interface

In the original SELinux kernel patch, the `open_secure` system call had two optional SID parameters, one to specify the SID of the file when a file is created and one to specify the SID of the file descriptor. However, calls such as `stat_secure` only returned the SID of the file, not the file descriptor, and no calls were provided to change the SID of an existing descriptor. For the LSM-based SELinux security module, file descriptors always inherit the SID of the opening process, and the `open_secure` system call only takes a single SID parameter to specify the SID of a new file. Hence, SIDs on file descriptors are completely invisible to applications, but are still used to control shared access to the file offset and flags.

5.3.6. 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 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. Storing Socket Security Data

The original SELinux kernel patch added security fields to the 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 kernel socket data structure, and SELinux uses this field to store security data for new connections during connection setup, when no user socket (and no inode) is available yet. However, the LSM-based SELinux security module stores the socket security data in the security field of the associated inode once the user socket is established. This is discussed further in Section 17 and Section 19.

5.4.2. 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. The SELinux network access controls required one additional hook (`sock_rcv_skb`) in two locations for controlling connection establishment and packet receipt on a socket. Another hook was added (`tcp_create_openreq_child` so that security data can be saved in the struct sock during connection establishment.

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. These three additional hooks have been accepted into the LSM kernel patch, so the LSM hooks are adequate for SELinux. The SELinux socket access controls are described in Section 17.3 and the SELinux network layer access controls are described in Section 19.

5.4.3. 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.

5.4.4. 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.

5.4.5. Extended Socket Calls

In the original SELinux kernel patch, a set of extended socket calls were implemented. The implementation of these calls for the LSM-based SELinux module is not yet complete and several unresolved issues still remain. A separate kernel configuration option has been defined for these calls and the corresponding hook function processing. Disabling this option has no impact on the enforcement of the network policy by the kernel, and no applications have been modified yet to use these calls, so the option can be disabled without harm. It is expected that further changes to the LSM kernel patch will be necessary to fully support the extended socket calls. The extended socket call processing is discussed further in Section 9 and Section 17.4.

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 5.1.6, 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, as described in Section 23.3. It also added new controls for some system operations that were not specifically addressed in the original SELinux kernel patch. New controls have been defined for `quotactl`, `syslog`, `swapon`, `nfsservctl`, and `bdflush`. These controls are discussed in Section 23. In the original SELinux kernel patch, these operations were merely controlled via the coarse-grained `capable` controls.

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 five major components: the security server, the access vector cache (AVC), the persistent label mapping, the new system calls, 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. 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. This code is largely unchanged from the original SELinux prototype, aside from some bug fixes, synchronization code, and preliminary devfs labeling support.

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 code can be found in the `avc.c` file. This code is also largely unchanged from the original SELinux prototype.

The persistent label mapping provides a mechanism for maintaining security contexts with persistent objects such as files and filesystems. It provides interfaces to the hook functions for getting and setting the security contexts for particular files. The persistent label mapping code can be found in the `psid.c` file. This code was derived from the original SELinux prototype, but was changed to store the inode-to-PSID mapping in a regular file rather than using a spare field in the on-disk inode, since LSM does not provide filesystem-specific hooks. This change should allow SELinux to support other filesystem types more easily, but has disadvantages in terms of performance and consistency. Additionally, several bug fixes were made and some new synchronization code was added.

The new system calls allow modified and new applications to be developed that have some degree of awareness of the new security features. One set of new calls is provided to allow applications to use the security server interfaces to obtain policy decisions for their own objects. The code for these calls can be found in the `ss/syscalls.c` file and is largely unchanged from the original SELinux prototype.

The other new system calls are typically extended forms of existing system calls that allow applications to obtain or specify security contexts for kernel objects or operations. The code for these calls can be found in the `syscalls.c` and `include/asm-i386/flask/syscalls.c` files. This code uses a different approach than the original SELinux prototype, which relied on the ability to generalize the existing internal kernel functions to support this functionality by directly patching it. The new code instead makes use of the existing system calls in combination with the ability to save state in the new security fields and the processing in the hook functions.

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 persistent label mapping 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 `selinux_plug.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 ext2 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 and Exit

This section describes the initialization and exit code for the SELinux security module. The initialization code is in the `selinux_plug_init` function in the `hooks.c` file. The exit code is in the `selinux_plug_exit` function in the same file.

7.1. selinux_plug_init

This function starts by initializing 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. 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.

If SELinux is built as a separate module (not recommended), the `security_init` is then called to initialize the security server and load the initial security policy configuration. If SELinux is built into the kernel, then the root filesystem has not been mounted yet, so the call to `security_init` is deferred to the `post_mountroot` hook in that case.

Next, the `selinux_plug_init` function inserts the `sys_security_selinux` function into the system call table in place of the LSM `sys_security` function. This is necessary to support the `execve_secure` system call, which requires access to the registers on the stack, as discussed in Section 9. 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_plug_exit

This function starts by calling the LSM `unregister_security` function to unregister the SELinux security module. It then restores the entry in the system call table used for `execve_secure`. Finally, it frees all of the security data structures associated with kernel objects. However, at present, this function does not free the memory associated with the AVC or the security server. Since these two components were permanently resident in the kernel in the original SELinux prototype, they do not currently provide interfaces for freeing their memory. This would not be difficult to add, but has not been a high priority since currently the SELinux module is built into the kernel.

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. 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. SELinux also provides some support for stacking with the owlsm security module, but only for options which do not require the use of the LSM security fields (i.e. not `CONFIG_OWLSM_FD`).

As mentioned in Section 7, the `selinux_plug_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, capabilities, and owlsm 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. Additionally, some of these hook functions are implemented in terms of the `capable` function, so stacking the `capable` hook is sufficient to cover them as well. However, there

would be no harm other than performance in always calling the secondary security module. The SELinux hook functions that call the secondary security module are:

- selinux_ptrace
- selinux_capget
- selinux_capset_check
- selinux_capset_set
- selinux_capable
- selinux_bprm_alloc_security
- selinux_bprm_set_security
- selinux_bprm_compute_creds
- selinux_task_post_setuid
- selinux_task_kmod_set_label
- selinux_inode_link
- selinux_inode_follow_link

More detail about these hook functions can be found in Section 23, Section 12, Section 11, and Section 14.

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. This has not yet been a priority.

9. New System Calls

This section discusses how the new SELinux system calls were implemented in the SELinux security module. The code for these calls can be found in the `syscalls.c` and `include/asm-i386/flask/syscalls.c` files. All of the new system calls are multiplexed through the `security` system call added by LSM. However, SELinux could not use the `sys_security` function and hook provided by LSM, because they do not provide access to the registers on the stack. This information is needed by the `execve_secure` system call.

Hence, the SELinux security module inserts its own `sys_security_selinux` function into the system call table during initialization in place of the LSM function. The SELinux function checks the module identifier to ensure that the application is invoking a SELinux system call and then calls the individual function for the requested call with the appropriate parameters. In the case of `execve_secure`, the entrypoint function also passes a pointer to the registers on the stack.

As mentioned in Section 6, the implementation of the extended system calls required a different approach than in the original SELinux prototype. Since the existing internal kernel functions could not be extended to pass SIDs, input and output SID arrays were added to the security structure associated with tasks (task_security_struct in `selinux_plug.h`). The extended system calls can set the elements of the *`in_sid`* array in this structure prior to calling the ordinary system call to pass SIDs to the hook

functions called during the system call. Likewise, the hook functions can set the elements of the *out_sid* array in this structure to pass SIDs back to the extended system calls for return to the application. Since a separate Linux task structure is created even when the `clone` call is used to create threads, these elements should be safe against concurrent access.

The new IPC system calls for obtaining SIDs were not as straightforward. The `semsid`, `shmsid`, and `msgsid` calls could not directly look up the corresponding kernel object due to the encapsulation of the IPC code, so they had to invoke an actual IPC operation to permit a hook to obtain the SID and pass it back via the *out_sid* array. The corresponding control operation (e.g. `SEMCTL`) is called with the `IPC_STAT` operation for this purpose, with a temporary kernel buffer and the data segment set to the kernel segment to deal with the normal copyout.

Similarly, the `msgrcv_secure` call was complicated by the fact that the `sys_msgrcv` function is not exported directly to modules and the generic `ipc` call expects a userspace ipc_kludge structure. This was resolved by using version `1` of the `MSGRCV` IPC call value, thereby avoiding the need to pass such a structure. In this case, it would not have worked to simply provide a temporary kernel structure and set the data segment, because the other parameters include userspace pointers.

The implementation of the extended socket system calls is still in progress, and several issues still remain to be resolved. These issues include passing a message SID and a destination socket SID for a particular outgoing message from the socket layer hooks to the network buffer hooks, and labeling the SYNACK packet with the correct SID when the useclient flag is set. These issues are discussed further in Section 17.4.

The final issue in implementing the new system calls was implementing the `execve_secure` call. As mentioned above, this call requires access to the registers on the stack, so SELinux had to provide its own entrypoint function for the `security` system call. This call parallels the processing of the existing kernel `sys_execve` entrypoint function, copying in the filename and calling the kernel `do_execve` function. It only differs in that it sets an element of the *in_sid* array to the specified SID for use by the program loading hook functions.

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 each SELinux security data structure defined in `selinux_plug.h`, the security 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 both by the precondition functions described in the next subsection and by the `alloc_security` and `free_security` hook functions.

Each primitive `alloc_security` helper function allocates a security structure of the appropriate type, sets a magic number field for subsequent sanity checking, sets a back pointer to the kernel data structure, adds the security structure to a list of similar structures, initializes the security information, and sets the object security field to refer to this new security structure. Currently, the security structure list and back pointer fields are only needed to deallocate and clear all security fields when the module exits. However, these lists and back pointers could also be useful in implementing revocation callback functions. Each

primitive `free_security` helper function clears the security field, removes the security structure from its list, and frees the security structure.

Since the `alloc_security` helper functions can be called from the precondition functions, they must synchronize the initial setting of the security field. To solve this problem, a spinlock is defined for each of these functions and used to synchronize access. Since precondition functions may also be invoked from interrupt context, the `alloc_security` helper functions use the `SAFE_ALLOC` flag for memory allocation and `spin_lock_irqsave` function for locking. The `SAFE_ALLOC` flag is defined in `include/linux/flask/flask_types.h`. This flag expands to `GFP_ATOMIC` if in interrupt context or to `GFP_KERNEL` otherwise.

10.2. Precondition Helper Functions

The SELinux security module defines a precondition function for each security structure (e.g. `task_precondition`, `inode_precondition`, etc). The SELinux hook functions invoke the appropriate precondition function on each kernel object prior to dereferencing its security field. If the security field is already set and the security structure is initialized, then the precondition function simply returns `1`, indicating that the hook can proceed. Otherwise, the precondition function attempts to allocate and/or initialize the security structure, returning `1` on success. If the precondition function returns a value less than or equal to zero, then the hook function immediately returns this value to its caller rather than proceeding to dereference the security field. A return value less then zero indicates an error and is a negative errno value as with other kernel functions. A return value of zero indicates that the security structure could not be initialized but the operation should proceed, e.g. during system initialization prior to the loading of the security policy or during the loading of the persistent label mapping for a filesystem.

The precondition functions serve several purposes. First, the precondition functions handle subjects and objects in the system that were created prior to module initialization. Some tasks and objects (e.g. the procfs root inode) are created prior to module initialization even when the module is compiled into the kernel, so there are always some pre-existing subjects and objects that must be handled. An alternative approach would be to traverse the kernel data structures (e.g. the task list and each task's open files) during module initialization and set the security field at that time for these pre-existing subjects and objects. However, locating all such subjects and objects may be difficult, especially if the module is dynamically loaded into a running kernel (e.g. an open file might be on a Unix domain socket awaiting receipt by a process). Hence, the precondition approach seems safer. Another alternative approach would be to view all such pre-existing subjects and objects as being outside the control of the module. However, this isn't an acceptable approach for a nondiscretionary access control scheme like SELinux.

It is important to note that the ability to determine the correct security attributes for these pre-existing subjects and objects may be limited. The SELinux module does what it can to determine the correct attributes after the fact, but it isn't always successful in the dynamically loaded module case. This is discussed in detail for inodes in Section 14.1.3 and for tasks in Section 11.1.3. We recommend always compiling the SELinux module into the kernel.

Second, the precondition functions handle objects whose security attributes cannot be fully determined at allocation time. For example, when an inode security structure is allocated, the `alloc_security` hook knows nothing useful about the inode, e.g. what kind of object will it represent (a file, a socket, a pipe, etc) and what specific object will it represent (for a file, what is the inode number or pathname?). All this hook can do is to mark the inode as unlabeled and save the label of the creating task for possible later use if the inode turns out to be a pipe or socket. If the inode is used to represent a file, then it will later be caught by the `post_lookup` hook, which can then set its security class and security identifier. If the

inode is used to represent a socket, then it will later be caught by the `post_create` hook or the `accept` hook, which can likewise set its security class and identifier. If the inode is used to represent a pipe, it may not be caught until it is actually used for a read or write. This issue could be avoided by providing an explicit hook in LSM for initializing pipe security attributes.

Third, the precondition functions serve to deal with cyclical dependencies. Such cycles can be created by dependencies between the module and the file system, e.g. loading the persistent label mapping for a file system or loading the security policy configuration.

10.3. Permission Checking Helper Functions

A set of helper functions on kernel objects and permissions are provided that invoke the appropriate precondition functions, dereference the security fields, 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. They also reduce the risk that a security field will be dereferenced without a call to the precondition function. 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 several 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 one of the new system 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. Third, some hook functions perform both a permission check and set an output SID for return to the application. In these latter two cases, 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 {
        unsigned long magic;
        struct task_struct *task;
        struct list_head list;
        security_id_t osid;
        security_id_t sid;
        security_id_t in_sid[2];
        security_id_t out_sid[2];
```

```
        avc_entry_ref_t avcr;
};
```

Table 1. task_security_struct

Field	Description
magic	Module id for the SELinux module.
task	Back pointer to the associated task_struct structure.
list	Pointer used to maintain the list of allocated task security structures.
osid	SID prior to the last execve.
sid	SID for the task.
in_sid[2]	Input SIDs used by SELinux system calls.
out_sid[2]	Output SIDs returned by SELinux system calls.
avcr	AVC entry reference.

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. task_precondition

This helper function is the precondition function for task security structures. This function ensures that the task security structure is allocated and initialized prior to use. If the task security structure is not already allocated, then the task was created prior to the loading of the SELinux module. In this case, this helper function attempts to retroactively determine the SID for the task.

If the task has no parent task, then this function assigns the `kernel` initial SID to the task. Otherwise, the security structure of the parent task is obtained and used to provide default values for the child task's security structure. The security structure for the inode that represents the task's executable is then obtained, and the SID of the task is computed based on the SID of the parent task and the SID of the inode using the `security_transition_sid` interface.

This parallels the computation that would occur normally if the parent task had forked the child and then the child had executed the program while running SELinux. However, there are several possible reasons why this computation might yield a different SID than the SID that would have been used if the SELinux module had been running when the child task was created. For example, the original parent task may have died or undergone a change in SID since creating the child. Additionally, if SELinux had been running at an earlier point, then the child task or one of its ancestors might have used one of the new system calls to explicitly set the SID, e.g. to set the user identity and role upon login.

11.1.4. selinux_task_kmod_set_label

This hook function is called by the kernel `exec_usermodehelper` function to set the security attributes for the kernel task running user-mode helper programs, such as modprobe. This is used for operations such as automatic kernel module loading and hotplug support. This hook function first calls the secondary security module to support Linux capabilities. It then sets the SID of the task to the `kmod` initial SID.

11.1.5. 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.

11.2. 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 security server system calls.

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	sigchld sigkill sigstop signal
selinux_task_wait	ChildTask	Current	sigchld sigkill sigstop signal

Only two 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. For all other signals, the generic `signal` permission is used.

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_setrlimit
- 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 or to provide policy control over resource limits.

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. selinux_bprm_alloc_security and selinux_bprm_free_security

The `selinux_bprm_alloc_security` and `selinux_bprm_free_security` hook functions currently do nothing for SELinux. At present, the SELinux module directly stores the new SID for the task in the security field of the linux_binprm structure, so a separate security structure is not allocated. In order to support stacking with other security modules that use the security field, the SELinux module will need to be changed to allocate a separate security structure and store the SID in this structure.

The `selinux_bprm_alloc_security` hook function calls the secondary security module to support the owlsm `RLIMIT_NPROC` check in this hook. However, since any use of the security field by the secondary module would create a conflict for the other SELinux binprm hooks, this hook also checks whether the secondary module set the security field. If so, then the secondary module is unregistered to prevent conflicts between SELinux and the secondary module on subsequent binprm hook calls. Stacking with such modules requires a common mechanism for chaining multiple security objects on the security field, as mentioned in Section 8.

12.1.2. 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 security field has already been set by a prior call, this hook merely returns. This allows security transitions to occur on scripts if permitted by the policy.

By default, this hook function sets the security field to the SID of the current task. This function checks the current task's security structure to see if the task specified a new SID for the task. If so, then this SID is used. 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.

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, 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 4. 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 4. 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 5.

Table 5. Permission Checks if Task SID is changing on exec

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

12.1.3. selinux_bprm_compute_creds

The `selinux_bprm_compute_creds` hook function is called to set the new security attributes for the task. This hook function first calls the secondary security module to support Linux capabilities. This hook then copies the current SID of the task into the old SID field of the task security structure to support the `getosecsid` system call. 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 is being traced, then the `ptrace` permission is checked between the parent task and the new SID. If the task was created via `clone` and has shared state, then the `share` permission is checked between the old and new SIDs. If these permission checks fail, then the task SID is left unchanged and the task is sent a SIGKILL to terminate it. These permission checks are shown in Table 6.

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

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

If all permissions are granted, this hook function changes the SID of the task to the new SID. It then calls the `flush_unauthorized_files` helper function to close any file descriptors to which the task should no longer have access. This 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. 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.

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 {
        unsigned long magic;
        struct super_block *sb;
        struct list_head list;
        security_id_t sid;
        struct psidtab *psidtab;
        unsigned char uses_psids;
        unsigned char initialized;
        unsigned long initializing;
        unsigned char uses_task;
        struct semaphore sem;
};
```

Table 7. superblock_security_struct

Field	Description
magic	Module id for the SELinux module.
sb	Back pointer to the associated superblock.
list	Pointer used to maintain the list of allocated superblock security structures.
sid	SID for the file system.
uses_psids	Flag indicating whether or not the file system uses persistent SIDs.
initialized	Flag indicating whether the security structure has been initialized.
initializing	Flag indicating whether the security structure is in the process of being initialized.
uses_task	Flag indicating whether inodes in this filesystem should inherit the SID of the creating task (e.g. pipes, sockets).
sem	Semaphore used to synchronize filesystem relabels.

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_precondition

This helper function is the precondition function for super_block security structures. This function ensures that the super_block security structure is allocated and initialized prior to use. If the filesystem can use the persistent label mapping, then the `psid_init` function is called to initialize the mapping and to set the SID of the super_block. This is used for regular persistent filesystem types like ext2 and reiserfs. If the filesystem is a pseudo filesystem for private objects such as pipes or sockets, then a flag is set to indicate that inodes associated with the filesystem should inherit the SID of the creating process. If the filesystem is a pseudo filesystem like procfs, devpts, tmpfs, or devfs, then an appropriate initial SID is assigned to the super_block.

13.1.4. selinux_post_mountroot

This hook function is called after the root filesystem has been mounted. If the security server has not yet been initialized, this function calls the `security_init` function to initialize the security server and load the initial policy configuration. A failure at this point is fatal unless the development module option is enabled, in which case SELinux will defer initialization and processing until a subsequent policy load or AVC toggle. If the security server has already been initialized (i.e. the hook has been called twice due to a `change_root` for an initrd), then this hook function tries to reload the policy from the new root filesystem. If the reload fails due to either a lack of permission for the current process or a lack of a policy on the new root filesystem, then SELinux will continue operating under the old (initrd) policy. The hook function then calls the `superblock_precondition` function on the root filesystem to initialize its persistent label mapping.

13.1.5. selinux_post_pivotroot

This hook function is called after a successful pivot of the root filesystem via the `pivot_root` system call, typically when an initrd is used. This hook function tries to reload the policy from the new root filesystem. If the reload fails due to either a lack of permission for the current process or a lack of a policy on the new root filesystem, then SELinux will continue operating under the old (initrd) policy.

13.1.6. selinux_post_addmount

This hook function is called after a non-root filesystem has been mounted. It calls `superblock_precondition` to initialize the persistent label mapping of the filesystem. However, this is obsoleted by the newer `selinux_check_sb` hook and will be reduced to doing nothing in the future.

13.1.7. selinux_post_remount

This hook function is called after a successful remount of a filesystem (i.e. after the mount flags have been changed). If the filesystem uses the persistent label mapping, then this hook calls the `psid_remount` function to update the mapping at this time if the filesystem was previously mounted read-only and is now mounted read-write.

13.1.8. selinux_umount_close

This hook function is called when a filesystem is being unmounted prior to checking whether the filesystem is busy. If the filesystem uses the persistent label mapping, then this hook calls the `psid_release` to free any memory and release any files used for the mapping.

13.1.9. selinux_umount_busy

This hook function is called when the kernel determines that a filesystem cannot be unmounted (e.g. the filesystem is busy) after calling the `umount_close` hook. If the filesystem uses the persistent label mapping, then this hook function calls `psid_init` to reinitialize the mapping.

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. It also saves the SID of the filesystem in an element of the *`out_sid`* array in the task security structure for use by the `statfs_secure` system calls.

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_check_sb

This hook function is called to check permission when mounting a filesystem after reading the superblock. This function checks `mount` permission between the current task and the filesystem. Prior to checking permission, `superblock_precondition` is called, so the persistent label mapping for the filesystem will be initialized by this hook.

13.2.5. 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.

13.2.6. selinux_pivotroot

This (recently added) hook function is called to check permission when pivoting the root filesystem. Since the `pivot_root` system call also invokes the `capable` function with the `CAP_SYS_ADMIN` capability, the SELinux module already requires that the current process have permission to use this capability. This hook enables security modules to impose finer-grained restrictions on the use of the operation, e.g. distinguishing the operation from other operations that use the same capability and basing decisions on the security attributes of the new root filesystem. The SELinux module does not yet take advantage of this ability, but a finer-grained permission check is planned for the future.

13.2.7. Summary of Filesystem Permission Checks

The permission checks for the super_block hooks are summarized in Table 8.

Table 8. Filesystem Permission Checks

Hook	Source	Target	Permission(s)
selinux_sb_statfs	Current	Filesystem	getattr
selinux_mount	Current Current	MountDirectory Filesystem	mounton remount
selinux_check_sb	Current	Filesystem	mount
selinux_umount	Current	Filesystem	unmount

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 ordinary filesystems like ext2 and reiserfs and pseudo filesystems like devfs, procfs, 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 {
        unsigned long magic;           /* magic number for this module */
        struct inode *inode;           /* back pointer to inode object */
        struct list_head list;         /* list of inode_security_struct */
```

```
        security_id_t task_sid;        /* SID of creating task */
        security_id_t sid;             /* SID of this object */
        security_class_t sclass;       /* security class of this object */
        avc_entry_ref_t avcr;          /* reference to object permissions */
        unsigned char initialized;     /* initialization flag */
        unsigned long initializing;    /* initializing flag */
        ctl_sid *ctl;
        struct semaphore sem;
};
```

Table 9. inode_security_struct

Field	Description
magic	Module id for the SELinux module.
inode	Back pointer to the associated inode.
list	Pointer used to maintain the list of allocated inode security structures.
task_sid	SID of the creating task.
sid	SID of the inode.
sclass	Security class of this inode.
avcr	AVC entry reference.
initialized	Flag indicating whether the security structure has been initialized.
initializing	Flag indicating whether the security structure is in the process of being initialized.
ctl	Pointer into the shadow sysctl table for /proc/sys entries (See Section 14.1.3.1).
sem	Semaphore for synchronizing file relabels.

When the extended socket call option is enabled, the inode_security_struct structure is extended to include additional fields related to the extended socket calls. This is discussed further in Section 17.4.

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` tries to save the SID of the current task in the *`task_sid`* field. If the security structure of the current task is not already set, this function merely sets this field to the unlabeled SID. The `selinux_inode_free_security` hook function merely calls the `inode_free_security` helper function.

The `inode_alloc_security` function can not safely call `task_precondition`, because `inode_alloc_security` may be called indirectly from `task_precondition`. Hence, callers of `inode_alloc_security` should first call `task_precondition` on the current task when possible. This is done by the `selinux_inode_alloc_security` hook function.

14.1.3. inode_precondition

This helper function is the precondition function for inode security structures. This function ensures that the inode security structure is allocated and initialized prior to use. Prior to initializing the inode security

structure, this function calls `superblock_precondition` to ensure that the security structure for the superblock that is associated with the inode has been allocated and initialized.

Since inodes can represent many different kinds of objects, the inode security class must be determined and set in the security structure. If the inode represents a socket, then the `socket_type_to_security_class` function is used to obtain the security class based on the socket family and type. The socket security classes are described in Section 17.2.1. Otherwise, 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 10. If the inode does not have any of the modes listed in Table 10, then it defaults to the file security class.

Table 10. 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

The inode security identifier (SID) is then determined based on information in the superblock security structure. If the filesystem can use the persistent label mapping, then the `psid_to_sid` function is called to obtain the SID of the inode. This is used for regular persistent filesystem types like ext2 and reiserfs.

If the inode represents a private object such as a socket or pipe, then the inode inherits the SID of the task that allocated its security structure. For inodes allocated after the initialization of the SELinux module, this is the same task that allocated the inode, so the private object inherits the SID of its creator. However, if the inode was allocated before the initialization of the SELinux module and subsequently caught by `inode_precondition`, then this may be a different task which simply happens to be the first to access the object since the module was loaded. Hence, for sockets and pipes, the principle of first use is applied to retroactively determine the SID of a pre-existing object. If SELinux is to be used as a separate module, then a better approach is needed for labeling pre-existing sockets and pipes.

The handling for pseudo filesystem types is specialized to provide reasonable security semantics for each type. At present, the SELinux security module defines labeling behaviors for the procfs, devpts, tmpfs, and devfs pseudo filesystem types. The handling for each of these filesystem types is described below.

14.1.3.1. Procfs File Labeling

For procfs inodes, the `procfs_set_sid` function is called to set the inode SID. The root directory inode is assigned the `proc` initial SID. Highly sensitive files such as `kmsg` and `kcore` are also assigned individual initial SIDs. The `sys` subdirectory and the per-process PID subdirectories are handled specially, as described below. Most inodes simply inherit the SID of their parent directory.

The `sys` subdirectory is assigned the `sysctl` initial SID. The SIDs of entries in the `sys` subdirectory are determined by traversing the ctl_sid_root_table hierarchical table. This table shadows the kernel sysctl

table and allows SIDs to be selectively assigned at any level, with unspecified entries simply inheriting the SID of the parent entry. Since a pointer into the table is saved in the parent inode's security structure, this function simply searches the parent inode's table and does not need to reconstruct an absolute pathname. This table is also used by the `sysctl` hook as described in Section 23.

The per-process PID subdirectories are assigned the SID of the associated process. For each top-level PID subdirectory, the task is looked up by PID and its SID is used for the inode. Entries within the PID subdirectories simply inherit the SID of their parent directory.

14.1.3.2. Devpts and Tmpfs File Labeling

For devpts and tmpfs inodes, a SID is assigned when the inode is first accessed based on the SID of the process and an initial SID defined for the filesystem type. This SID is computed using the `security_transition_sid` interface of the security server. This permits the security policy configuration to define derived types for each domain's pseudo terminal devices and for each domain's shared memory pseudo files via the type_transition rules.

14.1.3.3. Devfs File Labeling

For devfs inodes, a SID is assigned when the inode is first accessed based on the pathname (relative to the root of the filesystem) and the security class of the inode. This SID is computed using the `security_devfs_sid` interface of the security server. This permits the security policy configuration to define security contexts for devfs nodes based on their pathname. SELinux support for using devfs is still experimental.

14.1.4. selinux_inode_post_lookup

This hook function is called after a successful lookup. At this point, useful information such as the inode number and mode are available for use in determining the security attributes of the inode. This function simply calls the `inode_precondition` function to set the SID and security class on the inode if it has not already been set.

14.1.5. post_create

The `post_create` helper function is called by several inode post-operation hooks which are called after a successful file creation. This helper function sets information in the inode security structure for an inode that represents a newly created file. This function first tests whether the dentry for the newly created file has a null inode. This can happen if the filesystem did not instantiate the dentry for the new file, e.g. NFS does not instantiate a dentry for symbolic links. If the dentry has a null inode, then this function merely returns.

This function checks the current task's security structure to see if the task specified a SID for the new file. If so, then this SID is used. Otherwise, a SID is obtained from the security server by calling the `security_transition_sid` interface; passing in the task and directory SIDs. The `inode_security_set_sid` helper function is called to set the SID and security class in the inode security structure. This function then checks the superblock security structure to see whether the

filesystem uses a persistent label mapping. If so, then this functions call the `sid_to_psid` function to set the persistent SID for the inode in the persistent label mapping.

This function is called by the following inode hook functions:

- `selinux_inode_post_create`
- `selinux_inode_post_symlink`
- `selinux_inode_post_mkdir`
- `selinux_inode_post_mknod`

14.1.6. selinux_inode_post_link/rename

The `selinux_inode_post_link` hook function is called after a new hard link has been successfully created. The `selinux_inode_post_rename` hook function is called after a successful rename. Both of these hook functions immediately return. SELinux does not need to update any state when a new hard link is created or a rename occurs, because security attributes are associated with inodes, not pathnames.

14.1.7. selinux_inode_delete

This hook function is called when a deleted inode is released, i.e. an inode with no hard links has its use count drop to zero. The function calls the `clear_psid` to clear the persistent SID for the inode in the persistent label mapping.

14.1.8. selinux_inode_revalidate

This hook function is called to revalidate the inode attributes. When support for NFS file labeling is added to SELinux, this hook function will be used to revalidate the SID of the inode. At present, this hook function merely returns success.

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 two optional parameters. The first optional parameter, aeref, allows another AVC entry reference, such as the one in the file security structure, to be passed for use instead of the reference in the inode security structure. The second optional parameter, adp, allows other audit data, such as the particular dentry, to be passed for use if an audit message is generated. This function simply 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. This function saves the dentry in the audit data structure and 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 specified a SID for the new file. If so, 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 11.

Table 11. 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 12 and Table 13.

Table 12. Link Permission Checks

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

Table 13. 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 14, Table 15, and Table 16. The permissions in Table 14 are always checked. The permissions in Table 15 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 16 are only checked if the file is a directory and its parent directory is being changed by the rename.

Table 14. Basic Rename Permission Checks

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

Table 15. Additional Rename Permission Checks if NewFile Exists

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

Table 16. 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 Linux `permission` function to check permission when accessing an inode. It 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 17 specifies the SELinux permission that is checked for each permission mask flag when checking access to a directory. Table 18 provides the corresponding permission information when checking access to a non-directory file.

In Table 17, 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 17. Directory Permission Checks

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

In Table 18, 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 18. Non-Directory Permission Checks

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

14.2.7. 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 19.

Table 19. Remaining Inode Hook Permission Checks

Hook	Permission
selinux_inode_readlink	read
selinux_inode_follow_link	read
selinux_inode_setattr	setattr
selinux_inode_stat	getattr

Of these hooks, only two require further description. First, the `selinux_inode_setattr` hook merely checks the general `setattr` permission to the file. 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 nondiscretionary access control policies. Second, in addition to performing a permission check, the `selinux_inode_stat` saves the SID of the inode in an element of the *`out_sid`* array in the task security structure for use by the `stat_secure` system calls.

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 {
        unsigned long magic;
        struct file *file;
        struct list_head list;
        security_id_t sid;
        security_id_t fown_sid;
        avc_entry_ref_t avcr;
        avc_entry_ref_t inode_avcr;
};
```

Table 20. file_security_struct

Field	Description
magic	Module id for the SELinux module.
file	Back pointer to the associated file.
list	Pointer used to maintain the list of allocated file security structures.
sid	SID of the open file descriptor.
fown_sid	SID of the file owner; used for `SIGIO` events.
avcr	AVC entry reference for the file.
inode_avcr	AVC entry reference for the associated inode.

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` tries to associate the file with the SID of the current task. If the security structure of the current task is not already set, the file is associated with the unlabeled SID. Callers of this function should first call `task_precondition` on the current task if possible. The `file_free_security` simply releases all resources.

The `selinux_file_alloc_security` calls the `task_precondition` function to ensure that the SID of the current task is set and then calls the helper function. The `selinux_file_free_security` hook functions merely calls the helper function.

15.1.3. file_precondition

The `file_precondition` helper function ensures that the file security structure is allocated and initialized prior to use. This function calls `task_precondition` on the current task and then calls `file_alloc_security`.

15.1.4. 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 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 `dentry_has_perm` helper

function. In some cases (e.g. lseek), 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 `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_llseek

This hook function is called by the `lseek` and `llseek` system calls to control access to the file offset. It calls `file_has_perm` with no requested file permissions to simply check access to the file descriptor.

15.2.4. 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 21 shows the permission checks performed for each command.

Table 21. I/O Control Permission Checks

Command	Source	Target	Permission(s)
FIONREAD FIBMAP FIGETBSZ EXT2_IOC_GETFLAGS EXT2_IOC_GETVERSION	Current	FileDescriptor File	use getattr
EXT2_IOC_SETFLAGS EXT2_IOC_SETVERSION	Current	FileDescriptor File	use setattr
FIONBIO FIOASYNC	Current	FileDescriptor	use
Other	Current	FileDescriptor File	use ioctl

15.2.5. selinux_file_mmap

This hook function is called by `mmap` to check permission when mapping a file. At present, if anonymous memory is being mapped, i.e. the file parameter is `NULL`, no checks are performed. However, this may be changed later to ensure that execute access to anonymous memory can be controlled. If a file is being

mapped, then `file_has_perm` is called with a set of permissions based on the flags and protection parameters.

Since read access is always possible with file 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. However, on some architectures, read access to memory is sufficient to execute code from it, so the ability to strictly control code execution is limited on such architectures.

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.

15.2.6. selinux_file_mprotect

This hook function is called by the `mprotect` call to check the requested new protection for an existing mapping. This hook simply calls `selinux_file_mmap` with the file, new protection value, and the existing flags for the mapping.

15.2.7. selinux_file_lock

This hook function is called by the `flock` system call. 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 22.

Table 22. 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 always called from interrupt. It is passed the target task, a file owner structure and several other parameters that are unused by SELinux. 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.

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 {
        unsigned long magic;
        struct kern_ipc_perm *ipc_perm;
        security_class_t sclass;
        struct list_head list;
        security_id_t sid;
        avc_entry_ref_t avcr;
};
```

Table 23. ipc_security_struct

Field	Description
magic	Module id for the SELinux module.
ipc_perm	Back pointer to the associated kern_ipc_perm.
sclass	Security class for the IPC object (see Section 16.1.2).
list	Pointer used to maintain the list of allocated IPC security structures.
sid	SID for the IPC object.
avcr	AVC entry reference.

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

```
struct msg_security_struct {
        unsigned long magic;
        struct msg_msg *msg;
        struct list_head list;
        security_id_t sid;
        avc_entry_ref_t avcr;
};
```

Table 24. msg_security_struct

Field	Description
magic	Module id for the SELinux module.
msg	Back pointer to the associated IPC message;
list	Pointer used to maintain the list of allocated IPC message security structures.
sid	SID for the IPC message.
avcr	AVC entry reference.

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. If a SID was specified using one of the new IPC system calls, then the specified SID is used for the IPC object. Otherwise, the 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` function is called by the following allocation hook functions:

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

Each of these hook functions calls `task_precondition` on the current task prior to calling `ipc_alloc_security` to ensure that task security structure will be available. This is not handled within the primitive allocation function itself, as with the other primitive allocation functions, to ensure that no cycles arise, although this would not currently be a problem for IPC objects. These hook functions then 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 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.

16.1.4. ipc_precondition

This helper function is the precondition function for IPC security structures. This function ensures that the IPC security structure is allocated and initialized prior to use. If the security structure is not already allocated, then this function first calls `task_precondition` on the current task and then calls `ipc_alloc_security`. Since it cannot determine the appropriate security class automatically, the security class is passed by the caller.

16.1.5. msg_precondition

This helper function is the precondition function for individual message security structures. This function ensures that the message security structure is allocated and initialized prior to use. If the security structure is not already allocated, then this function simply calls `msg_msg_alloc_security`.

16.1.6. ipc_savesid

This helper function saves the SID of an IPC object in the *`out_sid`* array of the current task's security structure for use by the new system calls. This helper function first calls the appropriate precondition functions to ensure that the necessary security structures are available. This function is called by the IPC control hooks when the command is `IPC_STAT`.

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 calls the appropriate precondition functions and then calls the AVC to check whether the current task has a particular permission to an IPC object. The security class of the IPC object is passed by the caller in case the IPC object's security structure has not yet been allocated.

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 25. 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 25. ipc_permission Permission Checks

Flag	Permission
`S_IRUGO`	unix_read
`S_IWUGO`	unix_write

16.2.3. selinux_ipc_getinfo

When a task attempts to use a `*_INFO` command in a *ctl call on an IPC object, the kernel calls this hook function. This hook function checks the `ipc_info` system permission for the current task.

16.2.4. 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. If one of the new `*get_secure` system calls was used to specify a desired SID for the IPC object, then these hook functions also verify that the SID matches the desired SID.

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 26.

Table 26. Semaphore Control Permissions

Operation	Source	Target	Permission
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 27.

Table 27. Shared Memory Control Permissions

Operation	Source	Target	Permission

Operation	Source	Target	Permission
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 28.

Table 28. Message Queue Control Permissions

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 29. This hook function also sets the SID on the message if it is unlabeled. It uses the SID from the *`in_sid`* array of the task security structure if the new `msgsnd_secure` system call was used. Otherwise, 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 29. 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 30. In the case that the new `msgrcv_secure` system call is being used to specify a desired message SID, then this hook also checks the actual message SID against the desired message SID. This hook function also saves the SID of the message for use by the new system call. 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 30. 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 structures and perform access control for socket operations. This section describes these hooks and their helper functions. The section concludes by describing the optional hook function processing for the extended socket calls.

17.1. Socket Related Security Structures

Security information can be attached to two additional kernel objects, the kernel socket (struct sock) and the open request information block (struct open_request). The security fields attached to these objects are

used to reliably store the remote (peer) SID for a connection, and to label server sockets with the client SID when extended socket calls are used.

The sock_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.

```
struct sock_security_struct {
        unsigned long magic;           /* magic number for this module */
        struct sock *sk;               /* back pointer to sock object */
        struct list_head list;         /* list of sock_security_struct */
        security_id_t sid;             /* SID of the sock */
        security_id_t peer_sid;        /* SID of the network peer */
}
```

Table 31. sock_security_struct

Field	Description
magic	Module id for the SELinux module.
sk	Back pointer to the associated sock structure.
list	Pointer used to maintain the list of allocated sock security structures.
sid	SID of the socket; equal to user space socket SID.
peer_sid	SID of the peer socket.

The `socket_sock_alloc_security` and `socket_sock_free_security` hooks are used to allocate and free the security structure associated with the kernel socket. Security information is stored in the kernel socket in order to propagate the SID for a client to the user socket that is ultimately created on the server. However, because the new server socket is not created until the connection has been established, the SID for the client is stored in the kernel socket which is always present.

The kernel object, struct open_request, has an LSM security field as well. SELinux uses this field to store security information about the TCP client during connection establishment. See Section 17.4 for information on the definition of the open request security structure and it’s use.

17.2. Managing Socket Related Security Fields

The SELinux module uses the security structure for the inode associated with the user space socket, so the `inode_alloc_security`, `inode_free_security` and `inode_precondition` functions are also applicable to sockets. See Section 14.1 for a discussion of these functions. However, additional socket-specific hook functions are necessary to initialize and manage the information in these inode security structures for sockets. These hook functions are described below.

17.2.1. selinux_socket_post_create

After a socket structure has been successfully created, this hook function is called to update the inode security field with information that was not previously available. By default, the inode SID is set to the SID of the creating task. The socket object class is refined into separate object classes for the different types of sockets, as determined by the type and protocol family specified as parameters to the `socket`

system call. The security class is assigned according to Table 32. If the socket does not match any of the specified types, it defaults to the generic socket security class. The kernel socket (struct sock) associated with the socket will have it's SID set to the user socket SID. This SID is used to label outgoing packets from a socket that has no user space socket structure associated with it.

Table 32. Socket Security Classes

Protocol Family	Type	Security Class
PF_UNIX	SOCK_STREAM	unix_stream_socket
PF_UNIX	SOCK_DGRAM	unix_dgram_socket
PF_INET/PF_INET6	SOCK_STREAM	tcp_socket
PF_INET/PF_INET6	SOCK_DGRAM	udp_socket
PF_INET/PF_INET6	SOCK_RAW	rawip_socket
PF_NETLINK	*	netlink_socket
PF_PACKET	*	packet_socket
PF_KEY	*	key_socket

17.2.2. 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 17.3), this hook function sets the SID and security class for the new socket. The new socket always inherits the security class of the listening socket. By default, the new socket SID is initialized to the SID 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.

17.2.3. selinux_socket_post_accept

This hook function is called after calling the protocol family's `accept` function. This hook calls the `extsocket_post_accept` function (see Section 17.4).

17.2.4. selinux_tcp_connection_request

A new connection is being requested on a listening socket. This hook allows the LSM module to maintain security information about the client during the connection establishment. The only function performed by this hook is to call the `extsocket_tcp_connection_request` hook.

17.2.5. selinux_tcp_synack

A reply SYN/ACK is being sent for a connection request. This hook allows the LSM module to label the SYN/ACK packet. For SELinux, the label used will be the client SID or the listening socket SID, depending on the use of extended socket functionality. This hook is called after the `skb_set_owner_w`

hook, and therefore, will override any labeling done by that hook. The only function performed by this hook is to call the `extsocket_tcp_synack` hook.

17.2.6. selinux_tcp_create_openreq_child

This hook is called when a new TCP kernel socket is created, typically during the `accept` system call. The security data associated with the listening socket is preserved in the new kernel socket, and later used to label packets that are sent from the socket after the user space socket has been detached. After labeling the new socket, this hook calls the `extsocket_tcp_create_openreq_child` hook.

17.3. Controlling Socket Operations

17.3.1. socket_has_perm

This helper function checks whether a task has a particular permission to a socket. It first calls the precondition functions for the task and the socket's inode. It then calls the AVC to check the permission.

17.3.2. 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 (or inlining the logic of this function when the task and/or inode security structures are needed for additional processing). Some of the hook functions perform additional processing. The hook functions and the initial permission that they check are shown in Table 33. Any additional processing for the hook functions, excluding the optional extended socket call processing, is then described in the following subsections.

Table 33. 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

17.3.2.1. selinux_socket_bind

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.

17.3.2.2. selinux_socket_sendmsg

Prior to returning, this hook function calls the NSID hook `nsid_sock_sendmsg` to adjust the maximum segment size (MSS) for the IP packet to account for the IP options. See Section 20 for a description of the NSID functions.

17.3.3. selinux_socket_sock_rcv_skb (Transport Layer Hook)

This hook function is called by the transport layer network protocols (e.g. UDP, TCP, raw IP, etc) to control receipt of individual packets on a socket at a point where the destination socket and the receiving network device information is available. Unlike the previously discussed socket hook functions, this hook is passed a pointer to a kernel socket (sock) structure rather than a socket structure. This hook function must 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. If the socket is in a TCP `TIME_WAIT` state, then the sock structure pointer actually refers to a tcp_tw_bucket structure. The tcp_tw_bucket structure does not contain a *`socket`* field, so the *`socket`* field cannot be accessed in this case. In other cases, the *`socket`* field can be accessed but may be `NULL`, indicating that the socket has not yet been associated with an active user socket. In these cases, the hook function merely returns success. Further study of these cases is needed to determine whether this behavior is safe.

After obtaining the socket security information, the hook function must also obtain security information for the packet (network buffer). If no receiving network device is set for the packet, then the hook function merely returns success, since this implies that the communication is local and this hook function is not applicable. Otherwise, if the network buffer is still unlabeled, then this hook initializes the network buffer to the default message SID for the receiving network device. Normally, the network buffer is labeled during IP input processing, but an unlabeled network buffer might reach this hook if the kernel was configured without the LSM IP hooks or if SELinux was dynamically inserted into a running kernel with network buffers that had already been processed by the IP layer.

The hook function then checks `recvfrom` permission between the socket and the packet's source socket SID to control the receipt of the packet on the socket. Depending on the type and state of the socket and the kind of packet, additional processing may be performed. The additional processing is described below, and the additional permission checks are shown in Table 34. The optional extended socket call processing is described separately in Section 17.4.

If the socket is a TCP socket in the `TCP_LISTEN` state (server) and the packet has the `SYN` bit set, then the `acceptfrom` permission is checked between the listening socket SID and the packet's source socket SID (i.e. the client socket SID). If the socket is a TCP socket in the `TCP_SYN_SENT` state (client) and the packet has the `ACK` or `SYN` bits set (without the `RST` bit), then the `connectto` permission is checked between the client socket SID and packet's source socket SID (i.e. the server socket SID).

Table 34. Connection Establishment Permission Checks

Socket State	Packet Bits	Source	Target	Permission
TCP_LISTEN	SYN	ListeningSocket	ClientSocket	acceptfrom
TCP_SYN_SENT	ACK SYN	ClientSocket	ServerSocket	connectto

17.3.4. 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. 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 35.

Table 35. Unix Domain Permission Checks

Hook	Source	Target	Permission
unix_stream_connect	ClientSocket	ServerSocket	connectto
unix_may_send	SendingSocket	ReceivingSocket	sendto

17.4. Extended Socket Call Processing

The original SELinux kernel patch implemented a set of extended socket calls that could be used to specify and obtain SIDs for sockets, connections, and datagrams. The implementation of these calls for the LSM-based SELinux module is not yet complete and several unresolved issues still remain. The calls and their processing can be completely disabled via a separate kernel configuration option without any affect on the enforcement of the network policy by the kernel. No applications have been modified yet to use these calls, so they can be disabled without harm for now.

This section describes the current state of the extended socket call implementation in the SELinux module. The extended socket call processing is implemented within inline functions defined in the `extsocket.h` file. These functions are called by the appropriate hook functions. This section begins by describing the fields added to the inode security and open request structures to support the extended socket calls. It then describes each of the inline functions in `extsocket.h`.

17.4.1. Extended Inode Security Structure

When the extended socket call option is enabled, the inode_security_struct structure is extended to include additional fields related to the extended socket calls. The additional fields are defined as shown below.

```
        security_id_t msid;            /* SID of message on the socket */
        security_id_t dsid;            /* SID of desired destination socket */
        security_id_t peer_sid;        /* SID of the peer socket */
        security_id_t newconn_sid;     /* SID to use for new connections */
        int useclient;                 /* Use client SID for connections */
        access_vector_t conn_perm;     /* connection permission */
```

Table 36. Extended inode_security_struct

Field	Description
msid	Message SID.
dsid	Destination socket SID.
peer_sid	SID of the peer socket.
newconn_sid	SID for new connection sockets.
useclient	Flag indicating to use client SID for new connection sockets.
conn_perm	Connection permission for revalidation.

17.4.2. Open Request Security Structure

When the extended socket call option is enabled, the open_request_security_struct structure is available. This structure is used to store security information for during connection requests, before the new socket is created.

```
struct open_request_security_struct {
        unsigned long magic;           /* magic number for this module */
        struct open_request *req;      /* back pointer to open request object */
        struct list_head list;         /* list of open_request_security_struct*/
        security_id_t newconn_sid;     /* SID of the new connection */
};
```

Table 37. open_request_security_struct

Field	Description
magic	Module id for the SELinux module.
sk	Back pointer to the associated open_request structure.
list	Pointer used to maintain the list of allocated open_request security structures.
peer_sid	SID of the new connection; either the listening socket SID, or the client SID

17.4.3. Extended Socket Functions

The optional hook function processing for the extended socket calls is implemented in a set of inline functions in `extsocket.h`. Each function is described below.

17.4.3.1. extsocket_open_request_alloc_security

Allocate and initialize the open_request_security_struct security structure for the open request kernel object. Called by `selinux_open_request_alloc_security`.

17.4.3.2. extsocket_open_request_free_security

Free the open_request_security_struct security structure for the open request kernel object. Called by `selinux_open_request_free_security`.

17.4.3.3. extsocket_init

This function is called by `inode_alloc_security` to initialize the additional fields as necessary. The socket peer SID field is set to the `any_socket` initial SID.

17.4.3.4. extsocket_create

This function is called by `selinux_socket_create` and `selinux_socket_post_create` to obtain the SID for the new socket. If the `socket_secure` call was used, then the SID given in that call is returned. Otherwise, the SID of the creating task is returned. If the extended socket option is disabled, then this function always returns the SID of the creating task.

17.4.3.5. extsocket_connect

This function is called by `selinux_socket_connect`. If a particular destination socket SID was specified via the `connect_secure` call, then additional processing is performed. If the socket is an INET socket, then an additional `enforce_dest` permission check is performed between the destination socket SID and the destination node SID. This check ensures that the destination node is trusted to enforce the restriction on the destination socket. For all sockets, the destination socket SID is copied to the *`dsid`* field of the socket's inode in order to pass it to the `extsocket_skb_set_owner_w` function for labeling the outgoing packet. The peer SID of the socket is also set to the destination socket SID.

17.4.3.6. extsocket_listen

This function is called by `selinux_socket_listen`. If both a new connection SID and the useclient flag are set, then an error is returned. For non-stream sockets, use of the client SID is not supported, so an error is returned. Also, if the new connection SID is given and is not equal to the socket SID, and error is returned. Otherwise, the socket's use client flag is cleared and the new connection SID is set to the socket SID. No further processing is performed for non-stream sockets.

For stream sockets, if a new connection SID was specified via `listen_secure`, then an additional `newconn` permission check is performed between the socket SID and the new connection SID. The new connection SID is then copied into the socket's new connection SID. Otherwise, the socket new

connection SID is set to the SID of the socket. The use client flag is also copied into the socket's *useclient* field.

17.4.3.7. extsocket_accept

This function is called by `selinux_socket_accept` to set the connection permission of the new socket to `acceptfrom` for subsequent revalidation.

17.4.3.8. extsocket_post_accept

This function is called by `selinux_socket_post_accept`. The peer SID of the new connection socket is set to the peer SID field of the kernel socket. This field was set during `extsocket_tcp_create_openreq_child` for INET sockets, or during `extsocket_unix_stream_connect` for Unix sockets. If the listening socket's use client flag is set, then the SID of the new connection socket is changed to the peer SID, i.e. the client socket SID. The peer SID is also copied into the out SID array of the current task, so that it is accessible to the `accept_secure` system call and can be passed back to the application.

17.4.3.9. extsocket_sendmsg

This function is called by `selinux_socket_sendmsg`. If the socket is a stream socket, then this function verifies that the message SID and destination socket SID are valid if they were specified using the extended socket calls. For stream sockets, the message SID must equal the sending socket SID, and the destination socket SID must equal the peer SID. For TCP sockets, this function also revalidates the connection permission between the socket and its peer. For client sockets, the connection permission and the peer SID are set during connection establishment by `extsocket_sock_rcv_skb`. For server sockets, the connection permission is set by `extsocket_accept` and the peer SID is set by `extsocket_post_accept`.

If the socket is a non-stream socket and a message SID was specified, then `send_msg` permission is checked between the socket SID and the message SID. If the socket is a non-stream INET socket (e.g. UDP, raw IP), then this function also checks `sendto` permission between the socket and the destination socket SID. By default, the destination socket SID is set to the peer SID for the socket, which defaults to the `any_socket` initial SID unless specified by a prior `connect_secure` call. If a particular destination socket SID was specified via `send*_secure`, then the `enforce_dest` permission is checked between the destination socket SID and the destination node SID.

For all sockets, the destination socket SID, if specified, is copied to the *dsid* field of the socket's inode security structure in order to pass it to the `extsocket_skb_set_owner_w` function for labeling the outgoing packet. For non-stream sockets, the message SID is similarly copied to the *msid* field if it was specified.

[XXX Need to bind the (msid, dsid) pair to the particular message in some manner so that `extsocket_skb_set_owner_w` can ensure that it is only applied to the corresponding network buffers. Possibly maintain a list of (message identifier, msid, dsid) triples on the socket in the `extsocket_sendmsg` function that can be consumed by `extsocket_skb_set_owner_w`, but not clear how to identify the message uniquely and consistently across both functions. Possibly bind security data to struct msghdr via a security field or control data, but a security field would break application compatibility (msghdr is an exported structure) and control data may interfere with application-specified

control data. The original SELinux kernel patch required invasive changes to propagate the SIDs down to the skb allocation.]

17.4.3.10. extsocket_recvmsg

This function is called by `selinux_socket_recvmsg`. For stream sockets, this function copies the peer SID into both elements of the out SID array of the current task's security structure so that the `recv*_secure` calls can return this SID as the source socket SID and message SID to the application. For datagram sockets, the SIDs are copied from the individual datagram by the `extsocket_skb_recv_datagram` function.

17.4.3.11. extsocket_getsockname

This function is called by `selinux_socket_getsockname`. This function copies the socket SID into the out SID array of the current task's security structure so that the socket SID can be returned via the `getsockname_secure` extended system call.

17.4.3.12. extsocket_getpeername

This function is called by `selinux_socket_getpeername`. This function copies the peer socket SID into the out SID array of the current task's security structure so that the peer socket SID can be returned via the `getpeername_secure` extended system call. For client sockets, the peer SID are set during connection establishment by `extsocket_sock_rcv_skb`. For server sockets, the peer SID is set by `extsocket_post_accept`.

17.4.3.13. extsocket_sock_rcv_skb

This function is called by `selinux_sock_rcv_skb`. If the socket is a TCP socket in the `TCP_LISTEN` state (server socket) and the packet has the `SYN` bit set, then a connection is being requested, and several checks are performed. If the listening socket was set to use the client socket SID for new connection sockets (via a `listen_secure` call on the server), then the `newconn` permission is checked between the listening socket SID and the packet's source socket SID (i.e. the client socket SID) to ensure that the listening socket is allowed to create a new connection socket with the same SID as the client socket.

At this point, the new connection SID will be either the client SID (when the listening socket was set to use client) or the listening socket SID. Next, the `acceptfrom` permission is checked between the new connection SID and the SID of the packet.

If the packet's destination socket SID is set (due to a `connect_secure` call on the client) and this SID does not match the listening socket's new connection SID, the connection is refused. (XXX The listening socket's peer SID is set to the packet's source socket SID, but this will be overwritten by subsequent connections. This is unreliable.)

If the socket is a TCP socket in the `TCP_SYN_SENT` state (client socket), and the packet has the `ACK` or `SYN` bits set (without the `RST` bit), then the client is receiving connection acknowledgment from the server. Several checks are made and the peer SID is saved. If the socket's peer SID is set (via a `connect_secure` call) and this SID does not match the source socket SID of the packet, then the connection is reset. This check parallels the server-side check for the same condition. The client socket's

peer SID is set to the source socket SID of the packet, and the connection permission is set to `connectto` for subsequent revalidation.

If the TCP socket is in the `TCP_ESTABLISHED` state, then the connection permission (either `acceptfrom` or `connectto`) is revalidated so that policy changes can be reflected by the permission checks.

For non-stream sockets, if the packet's destination socket SID is set (via `send*_secure`) and it does not match the receiving socket's SID, then the packet is rejected. Likewise, if the receiving socket's peer SID has been set (via `connect_secure`), and it does not match the source socket SID of the packet, then the packet is rejected.

17.4.3.14. extsocket_tcp_connection_request

This hook is called by `selinux_tcp_connection_request`. The purpose of this hook is to set the new connection SID for the open request associated with the requested connection. If the listening socket is set to use the client SID on new connections, the new connection SID is set to the SID of the packet that initiated the connection request. In this manner the SID of the new server socket will be reliably set with the client SID when multiple connections are being established on a busy server socket. Otherwise, new connection SID is set to the listening socket's new connection SID.

17.4.3.15. extsocket_tcp_synack

This hook is called by `selinux_tcp_synack` to label the outgoing network packet for the SYN/ACK with the new connection SID taken from the open request structure. This SID was set by the `extsocket_tcp_connection_request` hook.

17.4.3.16. extsocket_tcp_create_openreq_child

This function is called by `selinux_tcp_create_openreq_child`. When the listening socket is set to use the client SID for new connections, this hook sets the SID of the newly created kernel socket to the SID from the open request structure. This SID is used to label outgoing packets from a socket that has no user space socket structure associated with it (as can happen during the socket shutdown operation). The hook also copies the SID of the network packet that established the connection into the kernel socket peer SID field. This peer SID is used by `extsocket_post_accept` to reliably set the peer SID of the user socket structure.

17.4.3.17. extsocket_unix_stream_connect

This function is called by `selinux_unix_stream_connect`. If the listening socket was set to use the client socket SID for new connection sockets (via a `listen_secure` call on the server), then the `newconn` permission is checked between the listening socket SID and the client socket SID to ensure that the listening socket is allowed to create a new connection socket with the same SID as the client socket. If the new connection SID does not match the listening socket SID, then the `connectto` permission is rechecked based on the new connection socket SID rather than the listening socket SID. If the destination socket SID is set (due to a `connect_secure` call on the client) and this SID does not match the new connection socket SID, then access is denied. The peer SID of the kernel socket associated with the new connection is set to the sending socket SID. This peer SID can be used by `extsocket_post_accept` to

reliably set the peer SID of the user socket structure. The connection permission is set for subsequent revalidation [XXX Revalidation for Unix stream traffic is not yet implemented].

17.4.3.18. extsocket_unix_may_send

This function is called by `selinux_unix_may_send`. If the receiving socket's peer SID has been set (via `connect_secure`), and it does not match the sending socket SID, then access is denied. Likewise, if the destination socket SID is set (via `send*_secure`) and it does not match the receiving socket's SID, then access is denied.

17.4.3.19. extsocket_skb_set_owner_w

This function is called by `selinux_skb_set_owner_w`. If the message SID (non-stream only) or destination socket SID are set for the socket, then these SIDs are copied into the network buffer and then cleared from the socket. These SID fields in the socket's inode security structure are set during `selinux_socket_sendmsg`. [XXX This is unreliable, see Section 17.4.3.9.]

17.4.3.20. extsocket_skb_recv_datagram

This function is called by `selinux_skb_recv_datagram`. This function copies the SIDs from a network buffer into the out SID array of the task security structure when a datagram is received by a task. This enables the extended socket calls to return these SIDs to applications.

18. Network Buffer Hook Functions

LSM provides a set of hooks for maintaining and propagating security information for network buffer structures (struct sk_buff). A security field was added to this structure, and the hooks provide methods for allocating, cloning, copying, and freeing this security field. The basic lifecycle hook functions are:

- `selinux_skb_alloc_security`: Allocates and assigns a security structure to a new network buffer.
- `selinux_skb_clone`: Sets the security field on a newly cloned buffer and increments the reference count.
- `selinux_skb_copy`: Copies the security structure to a newly copied buffer.
- `selinux_skb_free_security`: Decrements the reference count and, if zero, frees the security structure.

These basic hooks are not discussed further. In addition to these basic hooks, two other hooks are provided: `selinux_skb_set_owner_w` and `selinux_skb_recv_datagram`. The remainder of this section describes the network buffer security structure and the implementation for these two hooks.

18.1. Network Buffer Security Structure

The skb_security_struct structure contains security information for network buffers. This structure is defined as:

```
struct skb_security_struct {
        unsigned long magic;            /* magic number for this module */
        struct sk_buff *skb;            /* back pointer */
        struct list_head list;          /* list of skb_security_struct */
        __u8 opts;                      /* Bitmap of current options */
        __u8 mapped;                    /* Bitmap of mapped SIDs */
        __u8 invalid;                   /* Security state invalidated */
        atomic_t use;                   /* reference count */
        __u32 serial;                   /* Policy ID used to label datagram */
        security_id_t ssid;             /* Source SID */
        security_id_t msid;             /* Message SID  */
        security_id_t dsid;             /* Destination SID */
        void *data;                     /* Implementation specific data */
};
```

Table 38. skb_security_struct

Field	Description
magic	Module id for the SELinux module.
skb	Pointer to the SKB this structure belongs to.
list	Pointer used to maintain the list of allocated SKB security structures.
opts	Bitmap of flags indicating current packet labeling options.
mapped	Bitmap of flags indicating currently mapped remote SIDs.
invalid	Flag indicating that the security state of the SKB is invalid.
use	Reference count for the security structure.
serial	The policy serial number.
ssid	The SID of the source socket.
msid	The SID of the message; sockets that maintain message boundaries may label each message.
dsid	The desired SID of the destination socket.
data	Opaque pointer to data that may be associated with the SKB. Not currently used.

See Section 20, Section 19, and Section 17.3.3 for a discussion of how these fields are used by the labeled networking support, the IP hooks, and the `sock_rcv_skb` hook.

18.2. selinux_skb_set_owner_w

This hook sets the SID fields in a network buffer for an outgoing packet when the buffer is associated with a particular sending socket. The SID fields can then be used for permission checks and other processing related to the buffer. If labeled networking is used for the outgoing packet, then the SID fields are copied into the IP option by the `selopt_ip_label_output` function.

If the sending socket has no associated user socket, and the socket is a TCP socket, then the network buffer source and message SIDs are set to the kernel socket SID. Otherwise, no further determination is possible and the network buffer is left unlabeled.

If the sending socket has an associated user socket, but there is no inode security structure, then the network buffer' source and message SIDs are assigned either the TCP reset socket SID or the ICMP socket SID based on its family and protocol, and this hook returns. This logic handles kernel created sockets, since they are not caught by the LSM hooks.

Where there exists a inode for the socket, the source socket SID and message SID for the network buffer are set by default to the SID of the sending socket. However, the extended socket calls may change the SIDs used for the network buffer. See Section 17.4.3.19 for a discussion of the optional extended socket call processing.

18.3. selinux_skb_recv_datagram

This hook calls the `extsocket_skb_recv_datagram` function to perform the processing necessary for the extended socket calls.

19. IPv4 Networking Hook Functions

The SELinux IPv4 networking hook function implementations perform network layer access controls for outgoing and incoming packets. Many of these hooks are implemented by using the existing Linux kernel Netfilter framework, thereby minimizing the need for new hooks in the network protocol implementation. These hooks may use the security fields associated with network buffers (struct sk_buff), network devices (struct net_device), and sockets (the associated struct inode).

19.1. Netfilter-based Hook Functions

LSM allows a security module to intercept each Netfilter hook twice; both before and after the packets have passed through the standard kernel packet filtering mechanisms. Correspondingly, for each of the five types of NetFilter hooks, there are two LSM hooks registered. The hook name is suffixed with either `_first` or `_last` as appropriate. These hook functions follow the conventions of the Netfilter hooks rather than the conventions of other LSM hooks; hence, these hooks must return `NF_ACCEPT` to allow the packet through and `NF_DROP` to reject the packet.

19.1.1. selinux_ip_input_helper

This helper function is used by the `selinux_ip_preroute_last` and `selinux_ip_input_last` hooks to perform some functions common to those two hooks. This function takes as parameters the network buffer, the network buffer's security structure, and the receiving network device. If the network buffer is unlabeled, then this hook initializes the source socket SID and message SID to the default message SID of the receiving network device.

The hook function then uses the `security_node_sid` interface of the security server to obtain the SID associated with the source node (host) for the packet. It then checks a permission (based on the protocol type) between the network buffer and the receiving network interface and a permission between the

network buffer and the source node. The permission checked for each protocol type is shown in Table 39.

Table 39. Packet Receive Permissions

Protocol	Permission
UDP	udp_recv
TCP	tcp_recv
other	rawip_recv

19.1.2. selinux_ip_preroute_last

This hook function intercepts incoming packets after they have been received on the network interface, but prior to routing. Since it is called after any other Netfilter pre-routing hooks, packets may be modified or dropped prior to reaching this hook function. Since this hook function is a pre-routing hook, it is applied to packets that are not locally destined as well as those that are. The `selinux_ip_input_helper` function is called to initialize the network buffer SIDs and to check permissions for all received packets.

19.1.3. selinux_ip_input_first

This hook function intercepts incoming packets that are locally destined. It calls the NSID hook `nsid_ip_map_input` to map any remote SIDs saved in the network buffer security structure by `selinux_ip_decode_options` to local SIDs. See Section 20 for a description of the NSID functions.

19.1.4. selinux_ip_input_last

This hook function intercepts incoming locally destined packets after remote SIDs have been mapped. If the packet did not have a CIPSO label, then this hook does nothing, since all of the necessary processing was performed by `selinux_ip_preroute_last`. Otherwise, this hook function calls `selinux_ip_input_helper` again on the network buffer to recheck permissions based on the mapped SIDs.

19.1.5. selinux_ip_output_first

This hook function will return `NF_DROP` for packets labeled as invalid in the network buffer security structure. Otherwise, the `nsid_ip_label_output` hook is called to set the labels in the outgoing IP packet from the network buffer SIDs. See Section 20 for a description of the NSID functions. The result from this function call is returned by the hook.

19.1.6. selinux_ip_postroute_last

This hook intercepts outgoing packets after network routing, just before being put on the wire. Since it is called after any other Netfilter post-routing hooks, packets may be modified or dropped prior to reaching

this hook function. This hook function must obtain security information for the destination node (host). It uses the `security_node_sid` interface of the security server to obtain the SID associated with the destination node.

This hook function then checks a permission (based on the protocol type) between the network buffer and the sending network device. It also checks the same permission between the network buffer and the destination node. The permission checked for each protocol type is shown in Table 40. The SID used in these checks is the message SID stored in the network buffer security structure. In the case of forwarded packets, this SID was initialized by the `selinux_ip_preroute_last` hook during input processing. For locally generated packets, the `selinux_skb_set_owner_w` hook sets the message SID.

Table 40. Packet Send Permissions

Protocol	Permission
UDP	udp_send
TCP	tcp_send
other	rawip_send

19.1.7. Unused NetFilter-based Hooks

The SELinux security module does not currently use the remaining Netfilter-based hooks. The following list of hook functions simply return `NF_ACCEPT`:

- `selinux_ip_preroute_first`
- `selinux_ip_forward_first`
- `selinux_ip_forward_last`
- `selinux_ip_output_last`
- `selinux_ip_postroute_first`

19.2. IP Packet Lifecycle Hooks

A small number of additional hooks are provided for IP packet lifecycle events; they allow validation and propagation of security attributes at various times during IP packet processing. These hooks are called when IP packets are fragmented and defragmented, encapsulated and decapsulated, and when IP security options need to be processed. Since these hook calls are not implemented via Netfilter, they follow the conventions of the normal LSM hooks, returning `0` on success.

19.2.1. selinux_ip_fragment

This hook copies the network buffer security information from the existing buffer to the new buffer when the IP packet is being fragmented.

19.2.2. selinux_ip_defragment

This hook calls the NSID hook `nsid_ip_defragment` to handle any special processing needed when IP packets are defragmented. See Section 20 for a description of the NSID functions. The result of the call to `nsid_ip_defragment` is returned.

19.2.3. selinux_ip_decode_options

This hook function calls `nsid_ip_decode_options`. See Section 20 for a description of the NSID functions. The result of the call to `nsid_ip_decode_options` is returned by this hook.

19.2.4. Unused IP Packet Lifecycle Hooks

The SELinux security module does not currently use the remaining IP packet lifecycle hooks. The following list of hook functions simply return success:

- `selinux_ip_encapsulate`
- `selinux_ip_decapsulate`

20. Network Packet Labeling

SELinux can optionally be built with support for labeled networking via CIPSO/FIPS-188 IP Options. The Network SID (NSID) API provides a general framework for labeled networking for SELinux. Selopt is a particular implementation of this API that provides labeled networking for SELinux using CIPSO/FIPS-188 IP Options. The NSID and Selopt components were contributed to SELinux by James Morris. This section provides a brief discussion of the NSID API and Selopt, drawing from the existing documentation in [MorrisSeloptOverview2002].

20.1. NSID API

The Network SID (NSID) API provides a general framework for labeled networking that is intended to be independent of the underlying mechanism. The NSID interfaces called by SELinux are:

- `nsid_sock_sendmsg`: Adjust effective MSS for outgoing TCP data segments if necessary for network security labels. Called by `selinux_socket_sendmsg`.
- `nsid_ip_label_output`: Adds network security labels to outgoing packets based on the security structure of the associated network buffer. Called by `selinux_ip_output_first`.
- `nsid_ip_decode_options`: Decodes network security labels on incoming packets into the security structure of the associated network buffer. Called by `selinux_ip_decode_options`.
- `nsid_ip_map_input`: Maps remote security labels on incoming packets to local security labels. Called by `selinux_ip_input_first`.

- `nsid_ip_defragment`: Validates the security labels on incoming fragments so that the security information for a packet is consistent across the fragments.

The NSID component implements dummy operations for each of the NSID functions that provide the default implementations until a particular NSID implementation is registered via `nsid_register_ops`. The Selopt component registers its own operations during initialization, replacing these dummy operations.

20.2. Selopt

Selopt implements the NSID API using CIPSO/FIPS-188 IP options as the underlying mechanism for passing SIDs across the network. Selopt provides mechanisms for:

- Labeling IPv4 packets with local SIDs
- Specifying which packets require labeling
- Decoding labels from peers
- Mapping remote SIDs to local SIDs

Selopt adds the concept of a security perimeter to SELinux. A security perimeter is a group of trusted peers that have equivalent security policies. Security policies are equivalent if the security attribute spaces are identical and have the same meanings on each system. Hosts can be added to or removed from the perimeter at any time by using the **pt** utility. Outgoing packets to a host within the perimeter will be labeled. Incoming packets from a host within the perimeter must be labeled or they will be dropped. Labeled packets from hosts outside of the parameter will be dropped.

Since Selopt labels outgoing packets with local SIDs in the IP option and SIDs have only local meaning, a mapping mechanism is required to translate remote SIDs to local SIDs for incoming packets. To support such translation, a Security Context Mapping Protocol (SCMP) was defined that allows a peer to request a security context for a given SID. This protocol is described in [MorrisSCMP2001]. The security context can then be translated to a local SID by the local security server and stored in a network SID mapping cache. A daemon called **scmpd** implements the SCMP protocol.

Selopt defines up to three SIDs that can be included in the IP option. These SIDs are copied from the network buffer security structure for outgoing packets, and copied into the network buffer security structure for incoming packets. The complete list of Selopt security parameters is:

- `Bypass`: A flags indicating that the packet is implicitly labeled. The SCMP packets don’t have security labels and will have this flag set.
- `Serial`: 32-bit policy serial number
- `SSID`: 32-bit source socket SID
- `MSID`: 32-bit message SID
- `DSID`: 32-bit destination socket SID

20.2.1. selopt_ip_label_output

This function adds security labels to the IP packet by copying the SIDs from the network buffer security structure into the IP packet's options. However, if the packet destination is not in the perimeter, or is local, the packet is not labeled. The SSID is always set in the IP option. The MSID is only set if it differs from the SSID. The DSID is only set if it was specified.

20.2.2. selopt_ip_map_input

This function will return `NF_DROP` for any packet from outside the perimeter that is labeled, and for any unlabeled packet from within the perimeter. Otherwise, mapping of the packet SIDs is attempted.

Any packet that has the `Bypass` flag set in the options is accepted without mapping. Packets that have a local source address are also accepted without mapping. The packet SIDs are mapped by first checking the peer cache for a previous mapping (the "fast" path). If the cache lookup succeeds, then the packet is accepted. Otherwise, a map request is sent to the cache manager (the "slow" path), and `NF_QUEUE` is returned. In this case, the Netfilter logic will call the `selopt_queue_handler` funtion to queue the network buffer. When the reply message is received for the map request, Selopt will reinject the network buffer by calling the Netfilter function `nf_reinject`. Processing of the SKB will then continue on to the next Netfilter hook.

20.2.3. selopt_ip_decode_options

This function will decode the security labels from the options field of the IP packet header. For packets that are not being delivered to the local host, this function returns without decoding the options. Otherwise, the Selopt policy serial number, source SID, message SID, and destination SID are copied from the packet options field into the SKB security structure.

20.2.4. selopt_ip_defragment

This function is used verify security labels across IP fragments. At this time, labeled fragments are not supported, so this function prints a warning message to the system log and returns success.

20.2.5. selopt_sock_sendmsg

Before an IP packet with options can be sent out, the maximum segment size (MSS) must be adjusted. This function is called by the `selinux_socket_sendmsg` hook function to adjust the size of the MSS to account for the presence of Selopt security labels in the IP options field.

21. Network Device Hook Functions

The SELinux network device hook function implementations manage the security fields of network device structures (struct net_device). At present, LSM only provides a single hook function that is called when a network device is unregistered. The LSM project decided that it would be too invasive to provide hooks in all locations where network devices were probed or initialized. Hence, security modules are

expected to allocate and initialize the security field on the first access to the device. This section describes the network device hook and helper functions.

21.1. Managing Network Device Security Fields

21.1.1. Network Device Security Structure

The netdev_security_struct structure contains security information for network devices. This structure is defined as follows:

```
struct netdev_security_struct {
        unsigned long magic;
        struct net_device *dev;
        struct list_head list;
        security_id_t sid;
        security_id_t default_msg_sid;
        avc_entry_ref_t avcr;
};
```

Table 41. netdev_security_struct

Field	Description
magic	Module id for the SELinux module.
dev	Back pointer to the associated network device.
list	Pointer used to maintain the list of allocated network device security structures.
sid	SID for the network device.
default_msg_sid	SID used for unlabeled messages received on this network device.
avcr	AVC entry reference.

21.1.2. netdev_alloc_security and netdev_free_security

The `netdev_alloc_security` and `netdev_free_security` helper functions are the primitive allocation functions for network device security structures. These functions perform the usual processing for allocating and freeing security structures.

21.1.3. netdev_precondition

This helper function is the precondition function for network device security structures. If the network device security structure is not already allocated, this function calls `netdev_alloc_security` to allocate one. It then calls the `security_netif_sid` interface of the security server to obtain a device SID and a default packet SID for the network device. The default packet SID is used for incoming packets received on the network device unless a packet labeling mechanism was used. This precondition function is called by the IPv4 networking hook functions prior to accessing the network device security structure.

21.1.4. selinux_netdev_unregister

This hook function is called when a network device is unregistered. It calls `netdev_free_security` to free the security structure.

22. Module Hook Functions

At present, the SELinux module hook function implementations do nothing. Module operations are controlled by the security policy by limiting the use of the CAP_SYS_MODULE capability via the `selinux_capable` hook function. If finer-grained controls are later determined to be worthwhile (e.g. controls based on the actual name or content of the module), then additional access controls could be implemented in these hook functions. The hook functions are:

- `selinux_module_create_module`
- `selinux_module_init_module`
- `selinux_module_delete_module`

23. System Hook Functions

The remaining LSM hooks are defined directly in the top-level struct security_operations. Most of these hooks are used to control Linux system operations. This section describes the SELinux hook function implementations for these system hooks.

23.1. Capability-Related System Hook Functions

23.1.1. 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.

23.1.2. 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.

23.1.3. selinux_capset_check

This hook function is called by the kernel to check permission before setting the capability sets associated with a task or a set of tasks. It checks `capset` permission between the current and target tasks, and also calls the secondary module to permit it to perform any additional capability checking. However, this check is not always meaningful, since the target task is also set to current if a set of tasks was specified to the `capset` system call.

23.1.4. selinux_capset_set

This hook function is called by the kernel to set the capability sets associated with a task. It also checks `capset` permission between the current and target tasks since the target task may have been inaccurate in the `selinux_capset_check` hook function. It then calls the secondary module to set the capability sets, since SELinux does not maintain this information. SELinux does not perform any checks on the individual capabilities being set, since it revalidates each capability on use in the `selinux_capable` hook.

23.1.5. selinux_netlink_send

This hook function is called to save security information for a netlink message when the message is sent. The kernel `capable` function is called to check whether the current task (the sender) has the `CAP_NET_ADMIN` capability and the corresponding SELinux `net_admin` permission. If so, then this capability is raised in the effective capability set associated with the netlink message. Otherwise, the effective capability set is cleared.

23.1.6. selinux_netlink_recv

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

23.1.7. Summary of Capability-Related Permission Checks

Table 42. Capability-Related Permission Checks

Hook Function	Source	Target	Permission
selinux_capable	Task	Task	CapabilityPermission
selinux_capget	Current	TargetTask	getcap
selinux_capset_check	Current	TargetTask	setcap
selinux_capset_set	Current	TargetTask	setcap
selinux_netlink_send	Current	Current	net_admin

23.2. System Hook Functions that Defer to Capable

Some system operations are controlled by both the `capable` hook and a separate hook that offers finer-grained control. In many of these cases, the checking performed by `selinux_capable` is adequate for SELinux, so no other processing is required. Table 43 lists system hook functions for which no additional processing is required and the capability permission that is used to control the same operation. Of course, finer-grained permissions may be added to SELinux in the future, e.g. a permission to control what files can be used for accounting, so these hooks may be used at a later point in time.

Table 43. System Hook Functions that Defer to Capable

Hook	Permission Checked by Capable
selinux_sethostname selinux_setdomainname selinux_swapoff	sys_admin
selinux_reboot	sys_boot
selinux_ioperm selinux_iopl	sys_rawio
selinux_acct	sys_pacct

23.3. System Hook Function for sysctl

23.3.1. Shadow Sysctl Table

The ctl_sid structure is used to map a name from the sysctl namespace to a SID. This structure resembles the ctl_table structure defined in `sysctl.h`, with entries for the sysctl name (which is an integer), the associated string from the `/proc/sys` namespace, and the SID to be used for the entry. The last field is an optional pointer to a table containing the children of the entry.

A hierarchy of these tables is statically defined in the SELinux security module. Each level of the hierarchy is an array of ctl_sid entries. The layout corresponds to the hierarchy of ctl_table entries defined dynamically by the kernel and mapped into the `/proc/sys` file system. The hierarchy starts with the ctl_sid_root_table, providing SIDs for the top-level sysctl entries, and having several child tables. For example, the entry for `CTL_KERN` has a pointer to a table (ctl_sid_kern_table) for children of the `/proc/sys/kernel` entries.

23.3.2. search_ctl_sid

The `search_ctl_sid` helper function is used by the `selinux_sysctl` hook function to search the ctl_sid_root_table hierarchy for a SID corresponding to a given sysctl entry. The criteria used is that the ctl_name and procname must both match. Of course, this is only a heuristic and may not guarantee uniqueness. This function is recursive, and will return the SID corresponding to the ctl_sid table, or the `sysctl` initial SID if no match is found.

23.3.3. selinux_sysctl

This hook function checks permission for the current task to access a sysctl entry. It calls the `search_ctl_sid` helper function to obtain the SID associated with the sysctl entry. It 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 44 shows the permission checks associated with each requested operation.

Table 44. sysctl Permission Checks

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

23.3.4. Comparison with `/proc/sys`

The labeling of entries in `/proc/sys` by the `procfs_set_sid` function is described in Section 14.1.3.1. This function also uses the shadow sysctl table to determine SIDs for the inodes used to represent `/proc/sys` entries. These SIDs are then used in the file permission checks performed by the inode and file hook functions.

However, `procfs_set_sid` has certain advantages over `selinux_sysctl` in determining the SID of the sysctl entry. It can determine the parent inode of the entry, and it can save a pointer to the appropriate table in the inode's security structure. Hence, it only needs to search a single table, and can reliably identify the entry. Additionally, it can implement inheritance semantics so that the shadow table only needs to contain entries where the SID changes. To some extent, this could also be implemented in `selinux_sysctl` using the proc_dir_entry in the ctl_table. However, this would only work if procfs was enabled.

23.4. System Hook Function for 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.

23.5. System Hook Function for syslog

The `selinux_syslog` hook function checks that the current task has permission to perform a given system logging command. For operation `3`, 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.

23.6. System Hook Function for New System Calls

The `selinux_sys_security` hook function is called by the generic `security` system call, which is used as a multiplexor for new system calls for security-aware applications. However, since SELinux replaces the entrypoint function for the generic `security` system call, this hook is unused by SELinux. See Section 9 for further discussion.

23.7. Remaining System Hook Functions

Each of the remaining system hook functions performs a simple permission check, as summarized in Table 45. The `selinux_ptrace` hook function also calls the secondary module to permit it to perform additional capability checking.

Table 45. Remaining System Hook Function Permission Checks

Hook Function	Source	Target	Permission
selinux_ptrace	ParentTask	ChildTask	ptrace
selinux_swapon	Current	SwapFile	swapon
selinux_nfsservctl	Current	Kernel	nfsd_control
selinux_quotaon	Current	QuotaFile	quotaon
selinux_bdflush	Current	Kernel	bdflush

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.