Integrating Flexible Support for Security Policies into the Linux Operating System

Peter Loscocco, NSA, pal@epoch.ncsc.mil
Stephen Smalley, NAI Labs, sds@tislabs.com

February 2001

Contents

1 Introduction **2**

2 Overview **3**
- 2.1 Encapsulation of Security Policy 3
- 2.2 Flexibility in Labeling Decisions 4
- 2.3 Flexibility in Access Decisions 4
- 2.4 Support for Policy Changes 4
- 2.5 Process Controls 5
- 2.6 File Controls 5
- 2.7 Socket Controls 6

3 Security Server **7**
- 3.1 Architecture Types and Constants 7
- 3.2 Interfaces for the Kernel 7
- 3.3 System Calls for Applications 10
- 3.4 Policy Configuration Language 11
  - 3.4.1 TE configuration 12
  - 3.4.2 RBAC configuration 14
  - 3.4.3 MLS configuration 15
  - 3.4.4 User configuration 15
  - 3.4.5 Constraints configuration 16
  - 3.4.6 Security context configuration . . 16
- 3.5 Prototype Implementation 17
  - 3.5.1 compute_av 18
  - 3.5.2 compute_sid 18
  - 3.5.3 sid_to_context 19
  - 3.5.4 context_to_sid 19
  - 3.5.5 load_policy 19
  - 3.5.6 Other interfaces 20
  - 3.5.7 System Call Controls 20

4 Access Vector Cache **20**
- 4.1 Interfaces for the Kernel 20
- 4.2 Interfaces for the Security Server 23
- 4.3 Implementation 24

5 Process Management **25**
- 5.1 Design 25
  - 5.1.1 Object Classes 25
  - 5.1.2 Permissions 26
  - 5.1.3 Control Requirements 26
  - 5.1.4 API extensions 29
- 5.2 Implementation 30
  - 5.2.1 Labeling 30
  - 5.2.2 API Extensions 30
  - 5.2.3 Control Requirements 30

6 File System **32**
- 6.1 Design 32
  - 6.1.1 Object Classes 32
  - 6.1.2 Permissions 32
  - 6.1.3 Control Requirements 34
  - 6.1.4 Persistent Labeling 36
  - 6.1.5 API extensions 37

6.2 Implementation 37
6.2.1 Labeling 37
6.2.2 API extensions 38
6.2.3 Control Requirements 39

7 Other File System Types 41
7.1 Procfs 41
7.1.1 Procfs Analysis 41
7.1.2 Procfs Labeling Design 42
7.1.3 Procfs Labeling Implementation . 43
7.2 Devpts 43
7.3 NFS client support 44

8 Networking 44
8.1 Design 44
8.1.1 Object Classes 44
8.1.2 Permissions 45
8.1.3 Control Requirements 46
8.1.4 API extensions 48
8.2 Implementation 50
8.2.1 Labeling 50
8.2.2 API extensions 52
8.2.3 Control Requirements 52

9 System V IPC 55
9.1 Design 55
9.1.1 Object Classes 55
9.1.2 Permissions 55
9.1.3 Control Requirements 56
9.1.4 API extensions 57
9.2 Implementation 57
9.2.1 Labeling 57
9.2.2 API Extensions 57
9.2.3 Control Requirements 58

10 System Call Review 58
10.1 Process Management 58
10.1.1 Scheduling 58
10.1.2 Sessions and Process Groups . . . 58
10.1.3 User and Group Identity 59
10.1.4 Capabilities 59
10.1.5 Timers 59
10.1.6 Resource Limits and Usage . . . 59
10.1.7 Other Process Calls 59
10.2 Memory Management 60
10.3 File System 60
10.4 Kernel Modules 61
10.5 System Operations 61

11 To Do 62

Abstract

The protection mechanisms of current mainstream operating systems are inadequate to support confidentiality and integrity requirements for end systems. To address this problem, the National Security Agency (NSA) worked with Secure Computing Corporation (SCC) to develop a strong, flexible mandatory access control architecture based on Type Enforcement. The architecture, now called Flask, was prototyped in the Mach and Fluke research operating systems. The NSA is now integrating the Flask architecture into the Linux operating system to transfer the technology to a larger developer and user community. NAI Labs, SCC, and MITRE are assisting the NSA in this integration. This paper presents the design and implementation for integrating the security mechanisms of the Flask architecture into the Linux kernel.

1 Introduction

End systems must be able to enforce the separation of information based on confidentiality and integrity requirements to provide system security. Operating system security mechanisms are the foundation for ensuring such separation. Unfortunately, existing mainstream operating systems lack the critical security feature required for enforcing separation: mandatory access control [4]. As a consequence, application security mechanisms are vulnerable to tampering and bypass, and malicious or flawed applications can easily cause failures in system security.

To address this problem, the National Security Agency (NSA) worked with Secure Computing Corporation

(SCC) to research a strong, flexible mandatory access control architecture based on *Type Enforcement* [1], a mechanism first developed for the LOCK system [6]. The NSA and SCC developed two Mach-based prototypes of the architecture: DTMach [2] and DTOS [5]. The NSA and SCC then worked with the University of Utah's Flux research group to transfer the architecture to the Fluke research operating system. During the transfer, the architecture was enhanced to provide better support for dynamic security policies [8]. This enhanced architecture was named *Flask*. The NSA is now integrating the Flask architecture into the Linux operating system to transfer the technology to a larger developer and user community.

Researchers in the NSA's Information Assurance Research Office have implemented the architecture in the major subsystems of the Linux kernel, including mandatory access controls for operations on processes, files, and sockets. The Secure Execution Environments (SEE) group at NAI Labs is working with the NSA in further developing and configuring this security-enhanced Linux system. SCC and MITRE are assisting the NSA in developing application security policies and enhanced utility programs.

This paper describes work by the NSA and NAI Labs in integrating the security mechanisms of the Flask architecture into the Linux kernel. The paper begins by providing an overview of the Flask architecture and its Linux kernel implementation in Section 2. The design and implementation of two new operating system components, the security server and the access vector cache (AVC), are then described in detail in Section 3 and Section 4. Then, the design and implementation of security enhancements to each of the existing Linux operating system components are described in detail.

2 Overview

This section provides an overview of the Flask architecture and its Linux kernel implementation. It begins with a discussion of how the security policy is encapsulated in Flask. The section then discusses how Flask supports flexibility in labeling and access decisions. The ability of Flask to support policy changes is then described. It then describes the mandatory access controls provided for processes, files, and sockets.

2.1 Encapsulation of Security Policy

In the Flask architecture, the security policy logic is encapsulated within a separate component of the operating system with a general interface for obtaining security policy decisions. This separate component is referred to as the *security server* due to its origins as a user-space server running on a microkernel. In the Linux implementation, the security server is merely a kernel subsystem. The other kernel subsystems are referred to as *object managers* in the architecture.

The Flask architecture specifies the interfaces provided by the security server to object managers. The implementation of the security server, including any policy language it may support, are not specified by the architecture. The Linux implementation of the Flask security server defines a security policy that is a combination of Type Enforcement (TE), role-based access control (RBAC), and optionally multi-level security (MLS). The Linux security server has an associated policy language. A configuration written in this language is compiled by a separate program called `checkpolicy` into a binary representation read by the security server at boot time.

Since the content and format of security labels are dependent on the particular security policy, the Flask architecture defines two policy-independent data types for security labels: the security context and the security identifier. A security context is a variable-length string representation of the security label. Internally, the security server stores a security context as a structure using a private data type. A security identifier (SID) is an integer that is mapped by the security server to a security context. Flask object managers are responsible for binding security labels to their objects, so they bind SIDs to active kernel objects. The file system object manager must also maintain a persistent binding between files and security contexts. Since the object managers handle SIDs and security contexts opaquely, a change in the format or content of security labels does not require any changes to the object managers.

In the Linux implementation, a security context consists of a user identity, a role, a type, and optionally a MLS level or range. Roles are only relevant for processes, so file security contexts have a generic *object_r* role. The security server only provides SIDs for security contexts with legal combinations of user, role, type, and level or range. The individual attributes of the security context are not manipulated by the object managers.

2.2 Flexibility in Labeling Decisions

When a Flask object manager requires a label for a new object, it consults the security server to obtain a labeling decision based on the label of the creating subject, the label of a related object, and the class of the new object. For program execution, the Flask process manager obtains the label for the transformed process based on the current label of the process and the label of the program executable. For file creation, the Flask file system object manager obtains the label for the new file based on the label of the creating process, the label of the parent directory, and the kind of file being created. The security server may compute the new label based on these inputs and may also use other external information.

In the Linux implementation, the security server may be configured to automatically cause changes in the role or domain attributes of a process based on the role and domain of the process and the type of the program. By default, the role and domain of a process is not changed by program execution. The Linux security server may also be configured to use specified types for new files based on the domain of the process, the type of the parent directory, and the kind of file. A new file inherits the same type as its parent directory by default. For objects where there is only one relevant SID, object managers typically do not consult the security server. Instead, they merely use this SID as the SID for the new object. Pipes, file descriptions, and sockets inherit the SID of the creating process, and output messages inherit the SID of the sending socket.

2.3 Flexibility in Access Decisions

Object managers consult the Flask security server to obtain access decisions based on a pair of labels and an object class. The label pair is usually the label of a subject and the label of an object, but some access decisions may control relationships among object pairs. Each object class has a set of associated permissions. These permission sets are represented by a bitmap called an *access vector*. Flask defines a distinct permission for each service, and when a service accesses multiple objects, Flask defines a separate permission to control access to each object. For example, when a file is unlinked, Flask checks *remove_name* permission to the directory and *unlink* permission to the file.

The use of object classes in access requests allows distinct permission sets to be defined for each kind of object based on the particular services that are supported by the object. It also allows the security policy to make distinctions based on the kind of object, so that access to a device special file can be distinguished from access to a regular file and access to a raw IP socket can be distinguished from access to a UDP or TCP socket.

2.4 Support for Policy Changes

The Flask architecture includes an access vector cache (AVC) component that stores the access decision computations provided by the security server for subsequent use by the object managers. An object manager may further reduce the cost of a permission check by storing references to the appropriate entry in the AVC with its objects. As a result, most permission checks can occur without even incurring the cost of an extra function call.

The Flask AVC provides an interface to the security server for managing the cache as needed for policy changes. Sequence numbers are used to address the potential interleaving of access decision computations and policy change notifications. When the AVC receives a policy change notification, it updates its own state and then invokes callback functions registered by the object managers to update any permissions retained in the state of the object managers. For example, permissions may be retained in the access rights in page tables or in the flags on an open file description. After updating the state of the object managers and the state of the AVC to conform to the policy change, the AVC notifies the security server that the transition to the new policy has been completed.

In the Linux implementation of Flask, many permissions are revalidated on use, such as permissions for reading and writing files and permissions for communicating on an established connection. Consequently, policy changes for these permissions are automatically recognized and enforced without the need for object manager callbacks. Permissions can be efficiently revalidated by object managers using references to entries in the AVC. However, the revalidation of permissions on use is not adequate for revoking access to mapped file pages in the Linux page cache. The current implementation does invalidate the appropriate page cache entries when a file is relabeled, but a callback has not yet been defined to invalidate the appropriate page cache entries when a policy change notification is received.

The Linux implementation of the Flask security server provides an interface for changing the security policy configuration at runtime. The *security_load_policy* call may be used to read a new policy configuration from a file. After loading the new policy configuration, the security server updates its SID mapping, invalidating any SIDs that are no longer authorized, and resets the AVC. Subsequent permission checks on processes and objects with invalid SIDs always fail, preventing any further accesses by such processes and any further accesses to such objects. Support for automatically relabeling these processes and objects to a label that is accessible to administrators has not yet been implemented.

2.5 Process Controls

Flask provides several controls over the ability to change the label of a process. The security label of a process is only allowed to change upon program execution so that the inheritance of state and the initialization of the process in the new label can be controlled. Flask controls the ability of a process to transition to a new security label upon program execution through the *transition* permission, and it controls what programs may be used to perform such transitions through the *entrypoint* permission. It also controls the ability of a process to inherit open file descriptions across a transition.

Flask provides strong controls over the full set of code that can be executed by a process through the process *execute* permission. This permission is checked between the label of the transformed process and the label of the executable on every program execution. It is also checked when an ELF or script interpreter is executed, and when a file is memory-mapped with execute access (i.e. a shared library). This process *execute* permission differs from the separate *entrypoint* permission, which only controls what programs may be used to enter a new label. It also differs from the file *execute* permission, which only controls what programs may be initiated by a process, regardless of whether the process label is changed by the execution.

Flask controls the sending of signals, including the ability to indirectly send a signal via asynchronous I/O. It also controls the ability to trace another process, including the ability to continue tracing a process when a transition occurs. Flask controls several additional process management services, such as *fork*, *wait*, *setpgid*, *getpgid*, *getsid*, *setpriority*, *getpriority*, and the *sched* calls. These controls are described further in Section 5.1.

Flask provides an equivalent permission for each Linux capability. This allows the security policy to control the use of capabilities. Flask could be extended to provide a finer-grained replacement mechanism for capabilities. Such a mechanism was developed for one of Flask's predecessors, the DTOS system. This mechanism permitted privileges to be granted based on both the attributes of the process and the attributes of the relevant object, *e.g.* discretionary read override could be granted to a particular set of files. Since the mechanism obtained privilege decisions from the Flask security server, management of privileges was centralized and verification that privileges were granted appropriately was straightforward.

2.6 File Controls

Since open file descriptions may be inherited across *execve* or transferred through UNIX socket IPC, Flask labels and controls open file descriptions. An open file description is labeled with the SID of its creating process, since its state is usually treated as part of the private state of the process. It is important to distinguish between the label of an open file description and the label of the file it references. A read operation on a file changes the file offset in the open file description, so it may be necessary to prevent a process from reading a file using an open file description received or inherited from another process even though the process is allowed to directly open and read the file.

Flask labels file systems and controls services that manipulate file systems, including calls for mounting and unmounting file systems, the *statfs* call and the file creation calls. Flask controls the mounting of file systems through several permission checks. It requires that the process have *mounton* permission to the mount point directory and *mount* permission to the file system. It also requires that the *mountassociate* permission be granted between the root directory of the file system and the mount point directory. Flask does not yet perform any check between the device special file and the mount point.

Flask binds security labels to files and directories and controls access to them. Flask stores a persistent labeling table in each file system that specifies the security label for each file and directory in that file system. For efficient storage, Flask assigns an integer value referred to as a *persistent SID* (PSID) to each security label used by an object in a file system. The persistent labeling table is partitioned into a mapping between each PSID and its security label and a mapping between each object and its PSID. Since the table is stored in each file system, file labels are preserved if the file system is mounted at a different location or if the file system is moved to a different system.

In the Linux implementation, the mapping between each PSID and its security label is implemented using regular files in a fixed subdirectory of the root directory of each file system. This mapping is loaded into memory when the file system is mounted, and is updated both in memory and on the disk when a new security label is used for an object in the file system. The mapping between each object and its PSID is implemented by storing the PSID in an unused field of the on-disk inode. Since the PSID is available in the on-disk inode, no extra overhead is incurred either to obtain the PSID when a file is accessed or to set the PSID when a file is created. Additionally, since the mapping between each object and its PSID is inode-based, changes to the file system name space do not affect the mapping.

When an unlabeled file system is first mounted, a persistent labeling table is created for the file system, using a default label for all files obtained from the security server. Subsequently, existing files may be relabeled using new system calls. A program called `setfiles` is used to initially set file labels from a configuration file that specifies labels based on pathname regular expressions. This program and configuration file may also be used to reset file labels to a well-defined state. However, unless the configuration file is updated to reflect runtime changes in file labels, these changes will be lost when the program is executed. Runtime changes may occur as a result of new files being created, existing files being relabeled, or changes to the name space.

Flask provides a separate permission for each file and directory service. For example, Flask defines an *append* permission for files in addition to the *write* permission, and it defines separate *add_name* and *remove_name* permissions for directories to support append-only files and directories. Flask also defines a *reparent* permission for directories that controls whether the parent directory link can be changed by a *rename*.

Flask provides control over each object affected by a file or directory service. For example, in addition to checking access to the parent directory, Flask defines permissions for controlling access to the individual file itself for operations such as *stat*, *link*, *rename*, *unlink*, and *rmdir*.

2.7 Socket Controls

Flask provides control over socket IPC through a set of layered controls over sockets, messages, nodes, and network interfaces. At the socket layer, Flask controls the ability of processes to perform operations on sockets. At the transport layer, Flask controls the ability of sockets to communicate with other sockets. At the network layer, Flask controls the ability to send and receive messages on network interfaces, and it controls the ability to send messages to nodes and to receive messages from nodes. Flask also controls the ability of processes to configure network interfaces and to manipulate the kernel routing table.

Sockets effectively serve as communication proxies for processes in the Flask control model. Consequently, sockets are labeled with the label of the creating process by default. A process may create and use a socket with a different label to perform socket IPC with a different source security label. A process may set up a listening socket so that server sockets created by connections are labeled with either a specified label or with the label of the connecting client socket to act as a server for multiple labels.

Flask allows the security policy to distinguish between clients and servers for stream socket connections through the connectto and acceptfrom permissions. Flask allows the security policy to base decisions on the kind of socket through the use of object classes, and it allows the security policy to base decisions on the message protocol through the per-protocol node and network interface permissions.

Flask provides control over the association between INET domain sockets and port numbers and the association between UNIX domain sockets and files. Hence, the security policy can restrict the use of port numbers and pathnames for use by particular processes. Flask also provides control over open file description transfer via UNIX domain sockets.

In Flask, messages are associated with both the label of their sending socket and a separate message label. By default, this message label is the same as the sending socket label. A process may explicitly label individual messages if the underlying protocol supports message boundaries, i.e. datagram sockets. Messages sent on a stream socket all have the same label, which is the label of the stream socket.

Support for communicating message labels across the network has not yet been implemented in the Linux implementation of Flask. The Fluke implementation of Flask used IPSEC/ISAKMP both to label and protect messages, storing the labeling information in the IPSEC security association. During an ISAKMP negotiation, the appropriate security contexts are sent across the network and the peer obtains SIDs for these security contexts and stores them in its IPSEC security association. When messages are subsequently received that use the IPSEC security association, the messages are validated and then labeled with the SIDs from the association.

3 Security Server

The security server is a new operating system component that provides security policy decisions to the object managers, permitting the object managers to remain independent of the specific security policy that is used. This section describes the set of interfaces provided by the security server for kernel object managers and the set of system calls provided by the security server for security-aware applications. This section then describes the policy configuration language and the implementation of the current Linux security server prototype.

3.1 Architecture Types and Constants

The basic Flask types and constants are defined in the header file *include/linux/flask/flask_types.h*. The security context type (*security_context_t*) is defined as a string. The security identifier type (*security_id_t*) is defined as an unsigned 32-bit integer value. A null SID, *SECSID_NULL*, is defined to use when no particular SID is specified. A wildcard SID, *SECSID_WILD*, is defined that matches any other SID when used for certain access vector cache (AVC) operations. Certain SIDs (specified in *flask/initial_sids*) are predefined for system initialization. The corresponding constants are defined in the automatically generated header file *include/linux/flask/flask.h*.

The access vector type (*access_vector_t*) is defined as an unsigned 32-bit integer value. Each object class is identified by an unsigned 16-bit integer value, with the *security_class_t* type. The set of security classes is specified in *flask/security_classes*, with the corresponding constants in the automatically generated header file *include/linux/flask/flask.h*. The permissions for each security class are specified in *flask/access_vectors*, and the corresponding constants are defined in the automatically generated header file *include/linux/flask/av_permissions.h*.

3.2 Interfaces for the Kernel

The function prototypes for the security server interfaces provided for the kernel object managers are in the *include/linux/flask/security.h* header file. This subsection describes each of these interfaces. For each interface, the function prototype is listed followed by a description of the interface and a discussion of how the interface is currently used by the kernel object managers.

```
int security_init(void);
```

The *security_init* function initializes the security server. The kernel calls this function after the root file system is mounted (*fs/super.c:mount_root*) so that the security server may read configuration data from the root file system.

```
int security_compute_av(
        security_id_t ssid,
        security_id_t tsid,
```

```
        security_class_t tclass,
        access_vector_t requested,
        access_vector_t *allowed,
        access_vector_t *decided,
#ifdef CONFIG_FLASK_AUDIT
        access_vector_t *auditallow,
        access_vector_t *auditdeny,
#endif
#ifdef CONFIG_FLASK_NOTIFY
        access_vector_t *notify,
#endif
        __u32 *seqno);
```

The *security_compute_av* function computes access vectors based on a SID pair for the permissions in a particular class. The access vector cache (AVC) component calls this function when no valid entry exists for the requested permissions in the cache (*include/linux/flask/avc.h:avc_has_perm_ref_audit*). The first SID parameter, *ssid*, is referred to as the *source SID* and the second SID parameter, *tsid*, is referred to as the *target SID*. The returned access vectors must contain decisions for every permission specified in the *requested* access vector.

The security server may optionally return decisions for other permissions in the same class. The *decided* access vector contains the set of permissions for which a decision was returned. The other returned access vectors may only be used for permissions in this set. The security server may choose to defer computation of permissions until they are explicitly requested.

The *allowed* access vector contains the set of granted permissions. The *seqno* parameter contains a sequence number associated with the access granting. If the sequence number provided by the latest policy change is greater than this value, then the access granting may be invalid and must be discarded. The sequence number addresses the issue of an interleaving of an access granting and a policy change.

Two additional access vectors are returned if auditing support is enabled in the kernel configuration. The *auditallow* and *auditdeny* access vectors contain the set of permissions that should be audited when granted or when denied, respectively. These vectors enable the security server to precisely control the auditing of permission checks. The AVC component ensures that auditing is performed in accordance with these vectors (*avc_has_perm_ref_audit*).

One additional access vector is returned if notification support is enabled in the kernel configuration. The *notify* access vector contains the set of permissions for which the *security_notify_perm* function should be called when the operation associated with the permission has successfully completed. This vector permits the security server to request that the AVC component notify the security server of the successful completion of operations so that the security server may base its decisions on the history of operations in the system. This differs from merely basing decisions on the history of granted permissions, since an operation may still fail due to other conditions even if permission is granted for that operation. To support this functionality, the kernel object managers must be changed to notify the AVC component of the successful completion of operations by calling the *include/linux/flask/avc.h:avc_notify_perm_ref* inline function. The AVC component may then notify the security server if any of the *requested* permissions are in the corresponding *notify* vector. The necessary changes to the kernel object managers to notify the AVC component have not yet been implemented.

```
int security_notify_perm(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t requested);
```

The *security_notify_perm* function notifies the security server that an operation associated with the permissions in the *requested* access vector has completed successfully. The AVC component calls this function when it is called by an object manager to indicate that the operation has completed successfully if any of the *requested* permissions are in the corresponding *notify* vector (*include/linux/flask/avc.h:avc_notify_perm_ref*). Since the kernel object managers have not yet been changed to notify the AVC of operation completion, this function is currently never called.

```
int security_transition_sid(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
```

```
        security_id_t *out_sid);
```

The *security_transition_sid* function computes a SID for a new object based on a SID pair and a class. The kernel object managers call this function when objects are created if a SID was not specified for the object and there is more than one relevant SID that might be used as input in determining the SID of the new object. In particular, the file system code calls this function to obtain the SID of a new file based on the SID of the creating process and the SID of the parent directory, and the process management code calls this function to obtain the SID of a process transformed by an *execve* based on the current SID of the process and the SID of the executable program.

```
int security_member_sid(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        security_id_t *out_sid);
```

The *security_member_sid* function computes a SID to use when selecting a member of a polyinstantiated object based on a SID pair and a class. Certain fixed resources, such as the */tmp* directory or the TCP/UDP port number spaces, need be polyinstantiated to restrict sharing among processes. Each instantiation is referred to as a *member*. The kernel object managers call this function when a polyinstantiated object is accessed and then transparently redirect the process to the appropriate member. The necessary changes to the file system code and the networking code to support polyinstantiated directories and port number spaces are not yet implemented, so this function is currently never called.

```
int security_change_sid(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        security_id_t *out_sid);
```

The *security_change_sid* function computes a SID to use when relabeling an object based on a SID pair and a class. The `login` program will be modified to call this interface to obtain the SID to use when relabeling devices for a user session, based on the SID for the user session and the current SID of the device. This modification to `login` has not yet been implemented.

```
int security_sid_to_context(
        security_id_t  sid,
        security_context_t *scontext,
        __u32   *scontext_len);
```

The *security_sid_to_context* function returns the security context associated with a particular SID. The AVC component calls this function to obtain both security contexts for a SID pair when writing an audit record. The file system code calls this function to obtain a security context to use when adding an entry to the persistent label mapping. The *procfs* code calls this function to obtain the security context of a process to include in its *status* file. The *scontext* parameter is set to point to a dynamically-allocated string of the correct size. The *scontext_len* parameter is set to the length of the security context string, including the terminating *NULL* character. The string is allocated using *kmalloc* and must be freed with *kfree* by the caller.

```
int security_context_to_sid(
        security_context_t scontext,
        __u32   scontext_len,
        security_id_t *out_sid);
```

The *security_context_to_sid* function returns a SID associated with a particular security context. The file system code calls this function to obtain the SID that corresponds to a security context stored in the persistent label mapping. The *scontext_len* parameter specifies the length of the security context string, including the terminating *NULL* character.

```
int security_fs_sid(
        char *dev,
        security_id_t *fs_sid,
        security_id_t *file_sid);
```

The *security_fs_sid* function returns SIDs to use for an unlabeled file system mounted from the device specified by *dev*. The file system code calls this function when a process attempts to mount an unlabeled file system. The value for the *dev* parameter is a string of the form "*major:minor*" where both the major and minor number are in hexadecimal and are right justified in a two character field, as returned by the *kdevname* function on the device number. The *fs_sid* parameter is set to the SID to use for the file system, and the *file_sid* parameter is set to the SID to use for any existing files in the file system.

```
int security_port_sid(
        __u16 domain,
        __u16 type,
        __u8 protocol,
        __u16 port,
        security_id_t *out_sid);
```

The *security_port_sid* function returns the SID to use for the port number *port* in the protocol specified by the triple (*domain*, *type*, *protocol*). The networking code calls this function when a process attempts to bind a port outside of the range used to automatically bind sockets.

```
int security_netif_sid(
        char *name,
        security_id_t *if_sid,
        security_id_t *msg_sid);
```

The *security_netif_sid* function returns SIDs to use for a network interface. The networking code calls this function when a process first attempts to configure a network interface. The value for the *name* parameter is typically the driver name followed by the unit number, *e.g.* the name *eth0* would be used for the first Ethernet interface. The *if_sid* parameter is set to the SID to use for the interface, and the *msg_sid* parameter is set to the SID to use for any unlabeled messages received on the interface.

```
int security_node_sid(
        __u16 domain,
        void *addr,
        __u32 addrlen,
        security_id_t *out_sid);
```

The *security_node_sid* function returns the SID to use for the node whose address is specified by *addr*. The *addrlen* parameter specifies the length of the address in bytes, and the *domain* parameter specifies the communications domain or address family in which the address should be interpreted. The networking code calls this function when packets are sent to a node or received from a node.

```
int security_nfs_sid(
        __u16 domain,
        void *addr,
        __u32 addrlen,
        security_id_t *fs_sid,
        security_id_t *file_sid);
```

The *security_nfs_sid* function returns the SIDs to use for the file systems and files provided by the NFS server whose address is specified by *addr*. The *addrlen* parameter specifies the length of the address in bytes, and the *domain* parameter specifies the communications domain or address family in which the address should be interpreted. The NFS client code calls this function when a NFS file system is mounted.

3.3 System Calls for Applications

Security-aware applications in the system require the ability to convert between SIDs and security contexts in order to use the extended system calls provided by the kernel object managers. Furthermore, some security-aware applications act as object managers for the abstractions managed by the application, *e.g.* a windowing system acts as the object manager for its windows. These applications require access to most of the security server interfaces. Consequently, the security server provides a set of system calls for security-aware applications.

The function prototypes for the security server system calls used by applications are in the *include/ss.h* header file in the *syscalls* package. Currently, the security server system calls are implemented using a separate entry point for each call. All of the calls could alternatively be implemented using a single entry point, as is done for sockets with the *socketcall* system call. Since the security server system calls are derived from the interfaces provided to the kernel, this subsection compares and contrasts the system call interface with the interface for the kernel.

Several of the system calls are identical to the corresponding interface: *security_notify_perm*, *security_transition_sid*, *security_member_sid*, *security_change_sid* and *security_context_to_sid*. These calls are not discussed further in this subsection. There are currently no system calls that correspond to the interfaces that provide SIDs for kernel objects: *security_fs_sid*, *security_port_sid*, *security_netif_sid*, and *security_node_sid*. Calls could be added for these interfaces if applications require the same information.

Two of the system calls, *security_compute_av* and *security_sid_to_context*, differ from the corresponding interface only in the way in which their parameters are passed. For each of these system calls, the function prototype is listed below followed by a description of the call and a discussion of how the call should be used by applications.

```
struct security_query {
        security_id_t ssid;
        security_id_t tsid;
        security_class_t tclass;
        access_vector_t requested;
};

struct security_response {
        access_vector_t allowed;
        access_vector_t decided;
        access_vector_t auditallow;
        access_vector_t auditdeny;
        access_vector_t notify;
        __u32 seqno;
};

int security_compute_av(
    struct security_query *query,
    struct security_response *response);
```

The input parameters to the *security_compute_av* system call are provided in a *struct security_query* structure, and the output parameters are returned in a *struct security_response* structure. Like the kernel object managers, application object managers should use an AVC component to cache the results of a *security_compute_av* call. Like the kernel AVC component, the application AVC component must provide an interface to the security server for managing the cache as needed for policy changes. An application AVC component library has not yet been implemented.

```
int security_sid_to_context(
        security_id_t sid,
        security_context_t scontext,
        __u32 *scontext_len);
```

Whereas the *security_sid_to_context* interface for the kernel returns a dynamically-allocated string, the system call interface requires the application to provide a buffer for *scontext* and to initialize the *scontext_len* parameter with the size of the buffer. If the buffer is not large enough, then the *scontext_len* parameter is set to the correct length, `-1` is returned, and errno is set to *ENOSPC*. In this case, the application may allocate a buffer of the specified length and invoke the call again with the new buffer.

Two other system calls, *security_load_policy* and *security_get_sids*, do not have a corresponding interface used by kernel object managers. For each of these system calls, the function prototype is listed below followed by a description of the call.

```
int security_load_policy(
        char *path,
        __u32 pathlen);
```

The *security_load_policy* system call loads a new policy configuration from a file containing a binary representation of the configuration. The *pathlen* parameter specifies the length of *path*, including the terminating *NULL* character. If *pathlen* is zero, then the default policy configuration file is reloaded. After the new policy configuration has been loaded, the security server invokes the *avc_ss_reset* interface of the AVC component.

```
int security_get_sids(
        security_id_t *sids,
        __u32 *nel);
```

The *security_get_sids* system call returns the set of active SIDs. The *nel* parameter is initialized by the application to the number of elements in the array associated with the *sids* parameter, and modified on return to indicate the number of active SIDs. If the array is not large enough, then *nel* is set to the number of active SIDs, `-1` is returned, and *errno* is set to *ENOSPC*.

3.4 Policy Configuration Language

The current Linux security server prototype implements a security policy that is a combination of three subpolicies: type enforcement (TE), role-based access control (RBAC), and multi-level security (MLS). The MLS policy is only included if the *CONFIG_FLASK_MLS* kernel configuration option is enabled. The TE and RBAC policies are always included in the current implementation. This subsection describes the policy configuration language that may be used to customize these policies.

The policy configuration files are located in the *policy* directory. The *m4* macro processor is applied to these configuration files during the policy build, with the output written to *policy.conf*, and this output file is then compiled by the *checkpolicy* program into a binary representation stored in *policy*. The *policy* file is installed as */ss_policy*, and this file is read by the security server during initialization.

3.4.1 TE configuration The *all.te* file contains the configuration information for the type enforcement (TE) policy. This file is automatically generated from a collection of files. The *macros.te* file contains global macros used throughout the configuration for common groupings of classes and permissions and for common sets of rules. The *assert.te* file contains assertions that are checked after evaluating the entire TE configuration.

The *types* subdirectory contains several files with declarations for general types (types not associated with a particular domain) and some rules defining relationships among those types. Related types are grouped together into each file in this directory, e.g. all device type declarations are in the *device.te* file.

The *domains* subdirectory contains several subdirectories with a separate file containing the declarations and rules for each domain. Related domains are grouped together into each subdirectory, e.g. all domain definitions for system processes are in the *domains/system* subdirectory. The *domains/every.te* file contains rules that apply to every domain.

In a traditional TE policy, each subject is labeled with a domain, and each object is labeled with a type. A domain definition table (DDT) specifies the permissions granted between domains and types, and a domain interaction table (DIT) specifies the permissions granted between domain pairs. In contrast, the security server prototype merges the concept of domains and types into a single type abstraction, and maintains a single table that specifies the permissions granted between pairs of types. To the security server, a domain is simply a type that can be associated with a process. The TE configuration contains seven kinds of statements: type declarations, type transition rules, type member rules, type change rules, access vector rules, cloning rules, and access vector assertions.

```
type syslogd_t, domain;
type syslogd_exec_t, file_type,
     exec_type;
type devlog_t, file_type;
```

Figure 1: Type declarations.

A type declaration specifies a primary name for a type, an optional set of alias names, and an optional set of attributes. The primary name is always returned as the type by the *security_sid_to_context* call. The primary name or any of the alias names may be used for the type in a *security_context_to_sid* call. An attribute may be used to identify a set of types with similar properties. When an attribute name is used in a rule, it is expanded to the set of types with that attribute. Primary names, alias names, and attribute names all exist in the same name space.

Although domains are not distinguished from types by the security server, they can be optionally distinguished in the policy configuration using a “domain” attribute. A “domain” attribute has no intrinsic meaning to the security server, and is only meaningful to the extent the policy configuration uses the attribute in rules.

Sample type declarations for the *syslogd* daemon are shown in Figure 1. The *syslogd_t* type is the domain of the daemon process, so a *domain* attribute is associated with it. The *syslogd_exec_t* type is the type of the daemon program executable, so it is associated with both a *file_type* attribute and a *exec_type* attribute. The *devlog_t* type is the type of the */dev/log* socket file created by the daemon for receiving local log messages, so it is associated with a *file_type* attribute. These attributes have no intrinsic meaning to the security server, but they can be used in rules in the configuration.

A type transition rule specifies the default type of a transformed process or the default type of a new file based on the type of the creating subject and the type of a related object. For a process, the related object is the program executable. For a file, the related object is the parent directory. If no rule is specified, then the default type of a transformed process is the same as its type prior

```
type_transition
     initrc_t
     syslogd_exec_t:process
     syslogd_t;

type_transition
     syslogd_t
     device_t:sock_file
     devlog_t;
```

Figure 2: Type transition rules.

```
allow
    domain init_t:process
    sigchld;

allow
    syslogd_t device_t:dir
    { read getattr access search
      add_name remove_name };

allow
    syslogd_t devlog_t:sock_file
    create;
```

Figure 3: Access vector rules

to the *execve* call, and the default type of a new file is the same as the type of its parent directory. The default type is returned by the *security_transition_sid* call.

Each type transition rule has a creating subject type field, a related object type field, a class field, and a default type field. To permit concise specification of multiple type transitions, the type fields other than the default type field may contain a set of types, and the class field may contain a set of classes. An attribute name may be used to indicate the set of types with that attribute. A tilde may precede a set of types to indicate the complement of the set. An asterisk may be used to indicate all types. If multiple type transition rules are specified for a given type pair and class, then warnings are issued by the policy compiler and the last such rule is used.

Sample type transition rules for the *syslogd* daemon are shown in Figure 2. The first transition rule specifies that when an *rc* script executes the *syslogd* program executable, the transformed process should be assigned the *syslogd_t* type by default. The second transition rule specifies that when *syslogd* creates a socket file in */dev*, the socket file should be assigned the *devlog_t* type by default. The *device_t* type is the type assigned to the */dev* directory, and the *sock_file* class is the security class for socket files.

A type member rule specifies the type of the member object in a polyinstantiated object that should be accessed by default based on the type of the accessing process and the type of the polyinstantiated object. If no rule is specified, then the type of the polyinstantiated object is used for the member. A type member rule has the same syntax as a type transition rule except that it uses the *type_member* keyword.

A type change rule specifies the type to use for relabeling an object based on the type of the process and the current type of the object. If no rule is specified, then the type of the object is unchanged. A type change rule has the same syntax as a type transition rule except that it uses the *type_change* keyword.

An access vector rule specifies the permissions in an access vector based on a source type, a target type, and a class. There are four kinds of access vector rules: *allow*, *auditallow*, *auditdeny*, and *notify*. These rules define the corresponding access vectors returned by *security_compute_av*. If no rule is specified, then no permissions are returned in *allowed*, *auditallow*, or *notify*, and all permissions are returned in *auditdeny*. All permissions are always returned in the *decided* access vector, since the TE policy does not defer the computation of any permissions.

Each access vector rule has a source type field, a target type field, a class field, and a permissions field. As with type transition rules, sets of types and classes may be specified in the corresponding fields, and the asterisk and tilde characters may be used. Asterisk and tilde may also be used in the permission field. The name *self* may be used in the target type field to indicate that the rule should be applied between each source type and itself. If multiple access vector rules are specified for a given type pair and class, then the union of the permission fields is used.

Sample *allow* access vector rules are shown in Figure 3. The first rule grants every domain the ability to send a *SIGCHLD* signal to *init*, so that *init* can reap ev-

```
clone
    user_t sysadm_t;

neverallow
    syslogd_t
    ~{ syslogd_exec_t ld_so_t shlib_t }:process
    execute;
```

Figure 4: Domain cloning and access vector assertions.

```
role system_r types {
    init_t getty_t };
role user_r types user_t;
role sysadm_r types sysadm_t;
```

Figure 5: Role declarations

```
role_transition system_r login_exec_t login_r;
role_transition sysadm_r untrusted_exec_t user_r;
```

Figure 6: Role transition rules

ery process. The second rule grants *syslogd* the ability to access */dev* to replace */dev/log*. The third rule grants *syslogd* the ability to create the */dev/log* socket file.

A cloning rule specifies that the type transition and access vector rules defined for a specified source domain should be cloned for a specified target domain. A type transition rule for a process is not cloned if the default type in the rule is equal to the source or target domain in the clone statement. Hence, transitions from the source domain to itself or to the target domain are not cloned for the target domain. An access vector rule is not cloned if the target type in the rule is equal to the source or target domain in the clone statement. Hence, permissions between the source domain and itself or between the source domain and the target domain are not cloned for the target domain.

An access vector assertion specifies permissions that should not be in an access vector based on a source type, a target type, and a class. If any of the specified permissions are in the corresponding access vector, then the policy compiler will reject the policy configuration. Currently, there is only one kind of access vector assertion, *neverallow*, but support for the other kinds of vectors could be easily added. Access vector assertions use the same syntax as access vector rules.

A sample domain cloning rule and a sample access vector assertion are shown in Figure 4. The cloning rule clones the type transition rules and the access vector rules of the *user_t* domain for the *sysadm_t* domain. The access vector assertion rule verifies that the *syslogd* daemon process may only execute code from its program executable, the dynamic loader, and the system shared libraries.

3.4.2 RBAC configuration The *rbac* file contains the configuration information for the role-based access control (RBAC) policy. Although roles could be implemented directly using TE domains, this policy provides an additional layer of abstraction for grouping TE domains into roles and for expressing a role hierarchy. Roles are only relevant for processes. Files are labeled with a generic *object_r* role.

The RBAC configuration contains four kinds of statements: role declarations, role transition rules, role allow rules, and role dominance definitions. A role declaration specifies a name for the role and a set of types that may be associated with that role. This limits the set of types that may be entered by a process in the role. The generic *object_r* role may be associated with any type, since object roles are not relevant to the policy.

Sample role declarations are shown in Figure 5. The first declaration defines a *system_r* role for system processes such as *init* and *getty*. The second declaration defines a *user_r* role for ordinary users. The third declaration defines a *sysadm_r* role for system administrators.

A role transition rule specifies the default role of a transformed process based on its prior role and the type of the program executable. If no rule is specified, then the default role of a process is the same as its role prior to the *execve* call. Sample role transition rules are shown in Figure 6. The first rule specifies that when a system process executes the *login* program executable, the transformed process should be assigned the *login_r* role by default. The second rule specifies that when a system administrator executes an untrusted executable, the transformed process should be assigned the *user_r* role.

```
allow system_r user_r;
allow system_r sysadm_r;
allow system_r secadm_r;
```

Figure 7: Role allow rules

```
sensitivity unclassified alias u;
sensitivity top_secret alias ts;
dominance { u ts }
category nato;
category usuk;
level u;
level ts:nato,usuk;
```

Figure 8: MLS declarations

A role allow rule specifies allowable transitions between roles on an *execve*. If no rule is specified, then the change in roles will not be permitted. Additional controls over role transitions based on the type of the process may be specified through the *constraints* file, as discussed in Section 3.4.5. Sample role allow rules are shown in Figure 7. The first rule grants processes in the *system_r* role the permission to transition to the *user_r* role. The second and third rules provide similar permissions to the *sysadm_r* and *secadm_r* roles.

A role dominance definition specifies a hierarchy among a set of roles. A role automatically inherits any types that can be associated with any role it dominates in the hierarchy. As discussed in Section 3.4.5, this dominance relationship may also be used to define constraints on specific permissions. Role dominance definitions are not currently used in the sample policy configuration.

3.4.3 MLS configuration The *mls* file contains the configuration information for the multi-level security (MLS) policy. This policy is an extension of the Bell LaPadula (BLP) model of multi-level security in which each subject and object are labeled with a range of levels. If a subject is multi-level, *i.e.* its low level differs from its high level, then it is trusted to handle data at any level in its range while maintaining proper separation among the different levels. Multi-level objects may be used for the private state of multi-level subjects and for data sharing between multi-level subjects.

The MLS configuration begins by declaring the sensitivities and defining the dominance ordering for them. It then declares the categories, and defines levels by specifying what categories may be associated with each sensitivity. Sample MLS declarations are shown in Figure 8.

```
class tcp_socket {
    connectto  : { read write }
    acceptfrom : { readby writeby }
}
```

Figure 9: MLS base permissions

After the declarations, each access vector permission is mapped to a set of MLS base permissions (*read*, *write*, *readby*, and *writeby*). The *read* MLS base permission is only granted if the high level of the source SID dominates the high level of the target SID. The *write* MLS base permission is only granted if the target SID is single-level and it dominates the low level of the source SID, or if the range of the target SID is a subset of the range of the source SID. The latter restriction on writes to multi-level targets protects the integrity of such objects.

The *readby* and *writeby* MLS base permissions have the same requirements as the *read* and *write* MLS base permissions, respectively, with the source and target SIDs exchanged to reflect the target SID acting on the source SID. An access vector permission is only granted if all of the MLS base permissions associated with it are granted. Sample MLS base permission mappings are shown in Figure 9.

The current policy configuration language does not support specification of MLS range transition rules. A MLS range transition rule would specify the range of a new object based on the range of the creating subject and the range of a related object. By default, the MLS range of a process does not change across an *execve*, and the MLS range of an object is inherited from its creator.

The current policy configuration language also does not support specification of MLS range member rules for polyinstantiated objects. The MLS range of the member is currently always inherited from the process. Hence, a separate member is created for each distinct MLS range that accesses the object.

3.4.4 User configuration The *users* file, or the *users.mls* file if the MLS policy is enabled, contains one

```
user system_u roles system_r
    ranges u-ts;
user sds roles { secadm_r user_r }
    ranges { u s };
user pal roles user_r
    ranges { u s-ts };
```

Figure 10: User declarations.

or more declarations for users, as shown in Figure 10. Each user has a corresponding set of allowed roles that may be associated with that user. This limits the set of roles that may be entered by a process with that user identity. If the MLS policy is enabled, then each user also has a corresponding set of allowed MLS ranges that may be associated with the user. This limits the set of MLS ranges that may be entered by the user and the set of MLS ranges that may be used for objects owned by the user. Any MLS range that is a subset of one of the specified MLS ranges is allowed.

The current policy configuration language does not support specification of user transition rules. It is expected that the user identity of a process will only change through user authentication programs that explicitly specify the new identity. By default, the user identity of a process does not change across an *execve*, and the user owner of a file is inherited from the creating process. Controls over explicit user identity transitions based on the type of the process may be specified through the *constraints* file, as discussed in Section 3.4.5.

The current policy configuration language also does not support specification of user member rules for polyinstantiated objects. The user owner of the member is currently always inherited from the polyinstantiated object. Hence, separate members are not created for different users of processes that access the object.

3.4.5 Constraints configuration The *constraints* file defines additional constraints on permissions in the form of boolean expressions that must be satisfied in order for specified permissions to be granted. These constraints are used to further refine the type enforcement tables and role allow rules. Constraints may compare the user identity, role, or type of the source and target SIDs. Constraints may also compare the user identity, role, or type of either SID against a set of specified users, roles or types. Role comparisons may also be based on any dominance hierarchies defined in the RBAC configuration.

```
constrain process transition
    ( u1 == u2 or
      t1 == privuser );

constrain process transition
    ( r1 == r2 or
      t1 == privrole );
```

Figure 11: Process transition constraints.

```
sid kernel system_u:system_r:kernel_t:u
sid init system_u:system_r:init_t:u
sid kmod system_u:system_r:kmod_t:u
```

Figure 12: Security contexts for initial SIDs.

Sample constraints for changes in user identity and role for processes are shown in Figure 11. The first constraint requires that the user identity remain the same across an *execve* unless the process is in a type with the "privuser" attribute. *u1* and *u2* refer to the user identities of the source and target SIDs, respectively. *t1* refers to the type of the source SID. The "privuser" attribute would typically be limited to the domain for *login*.

The second constraint requires that the role remain the same across an *execve* unless the process is in a type with the "privrole" attribute. *r1* and *r2* refer to the roles of the source and target SIDs, respectively. This constraint is in addition to the requirement that any role change be authorized by a role allow rule in the RBAC configuration. The "privrole" attribute would typically be limited to the domain for *login*. It might also be associated with the domain for a *newrole* program to allow users to change roles within a session.

3.4.6 Security context configuration The *initial_sid_contexts* file, or the *initial_sid_contexts.mls* file if the MLS policy is enabled, contains the security context for each SID that was predefined for system initialization. Each security context consists of a user, a role, a type and, if the MLS policy is enabled, a MLS range, as shown in Figure 12. Since the initial SIDs do not correspond to authenticated users, they use a

```
3 2 system_u:object_r:public_t:u
    system_u:object_r:public_t:u
```

Figure 13: Security contexts for unlabeled filesystems.

```
tcp 21 system_u:object_r:ftp_t:u

eth0 system_u:object_r:netif_eth0_t:u
     system_u:object_r:netmsg_eth0_t:u

127.0.0.1 255.255.255.255
          system_u:object_r:node_lo_t:u

10.33.1.2 255.255.255.255
      system_u:object_r:nfs_clipper_t:u
      system_u:object_r:nfs_clipper_t:u
```

Figure 14: Security contexts for network objects.

system_u user identity.

The *fs_contexts* file, or the *fs_contexts.mls* file if the MLS policy is enabled, contains the security contexts to use when an unlabeled file system is mounted from a device, as shown in Figure 13. For each file system, the major and minor device numbers of the device are specified, followed by the file system security context and the security context for existing files in the file system. If no entry is specified for a device, then the security contexts associated with the initial SIDs *fs* and *file* are used. These initial SIDs are also used for the root file system if it is unlabeled, since the security server is not yet initialized when the root file system is mounted.

The *net_contexts* file, or the *net_contexts.mls* file if the MLS policy is enabled, contains the security contexts for port numbers, network interfaces nodes, and NFS servers, as shown in Figure 14. The current policy configuration language only supports ports and addresses in the *AF_INET* address family, although the security server interfaces are more general. For each port, the protocol (*tcp* or *udp*) and port range are specified followed by the port security context. If no entry is specified for a port, then the security context associated with the initial SID *port* is used.

For each network interface, the interface name is specified followed by the interface security context and the security context for any unlabeled messages received on the interface. If no entry is specified for a network interface, then the security contexts associated with the initial SIDs *netif* and *netmsg* are used. For each node, a network address and a network mask are specified, followed by the node security context. The mask is applied to the node address passed to the *security_node_sid* interface, and the result is then compared to the network address. In the current implementation, the entries are checked for a match in the same order that they are specified in the configuration. If no matching entry is specified for a node, then the security context associated with the initial SID *node* is used.

For each NFS server, a network address and a network mask are specified, followed by the file system security context and file security context. The mask is applied to the node address passed to the *security_nfs_sid* interface, and the result is then compared to the network address. In the current implementation, the entries are checked for a match in the same order that they are specified in the configuration. If no matching entry is specified for a node, then the security context associated with the initial SID *nfs* is used.

3.5 Prototype Implementation

This section describes the implementation of the current Linux security server prototype. The security server source code is located in the *security* subdirectory. In addition to being used to build the security server, this code is used in combination with *policy_scan.l* and *policy_parse.y* to build the *checkpolicy* program. The *checkpolicy* program is used to compile the policy configuration data into a binary representation for the security server. The *-d* option to the *checkpolicy* program may be used to interactively test the security server functions on a policy configuration. This option permits testing of a policy configuration prior to loading it into a running security server or prior to booting a kernel with it.

The *security_init* interface of the security server is implemented in *init.c*. This function calls *policydb.c:policydb_read* to create an in-memory representation of the policy configuration data (*policydb.h:policydb_t*) from the */ss_policy* file. It then calls *policydb.c:policydb_load_isids* to load the initial SIDs from the policy configuration into the SID table (*sidtab.h:sidtab_t*). Finally, it sets a global flag, *ss_initialized*, to indicate that the security server has initialized.

All of the other interfaces of the security server are implemented in *services.c*, with the corresponding system call functions in *syscalls.c*. Each of the interface functions disables interrupts locally and takes a single global spin lock (*ss_lock*) on entry using *spin_lock_irqsave*, and each function uses *spin_unlock_irqrestore* before returning. This locking scheme is likely to change to one that uses reader-writer spinlocks, since several of the security server functions only require read access to its global data structures. Several of these functions are split into a small stub function that handles locking and a separate function with the prefix *unlocked_* that implements the interface functionality. Although this separation is not currently used, it could be used to permit the security server to call one of the *unlocked_* functions from another function without doubly locking.

3.5.1 compute_av

The *unlocked_security_compute_av* function sets the sequence number to the value of *latest_granting*, a global counter that is incremented by the *security_load_policy* function when a new policy configuration is loaded. Then, the function sets the *decided* vector to contain all permissions, since none of the policies implemented by the security server prototype defer the computation of any permissions.

If the security server has not yet initialized, then the *unlocked_security_compute_av* function simply returns the *requested* permissions in both the *decided* vector and the *allowed* vector. Hence, all requested permissions are granted until the initialization of the security server has completed. This is necessary because some permission checks occur before the *security_init* function is called, *e.g. fork* permission for kernel threads and *create* permission for the ICMP socket and the TCP reset socket. Additionally, a *search* permission check occurs when the *security_init* function opens the policy configuration file.

A more secure solution would be to preload the security server state or the access vector cache state with the exact set of permissions that are required to initialize. This initial state could then be included in the analysis of the overall security policy. However, since the system is still under development, the full initial state is not yet known.

If the security server has initialized, then the *unlocked_security_compute_av* function looks up the security contexts for the SID pair in the SID hash table (*sidtab.h:sidtab_t*). These security contexts are stored using a structure that is private to the security server (*context.h:context_struct_t*). The function then looks up the attributes associated with the class (*policydb.h:class_datum_t*).

The function sets the values of the access vectors to their default values. It then looks for an access vector rule in the TE access vector table (*avtab.h:avtab_t*) for the type pair and class. If a rule exists, then the function sets the corresponding access vectors to the vectors specified by the rule.

If the MLS policy is enabled, the function then calls the MLS policy (*mls.c:mls_compute_av*) to remove any permissions from the *allowed* vector that are prohibited by the MLS policy. The MLS policy removes any permissions from *allowed* that are mapped to a MLS base permission that would be denied.

The function then checks the list of constraints associated with the class for any constraints that apply to the permissions in *allowed*. The *constraint_expr_eval* function is invoked on each such constraint. If the constraint evaluates to false, then the function removes the constrained permissions from *allowed*.

If the process transition permission is being computed and the role is changing, then the function looks for a role allow rule that authorizes the role transition. If no such rule exists, then the process transition permission is denied.

3.5.2 compute_sid

The *unlocked_security_compute_sid* function is used for the *security_transition_sid*, *security_member_sid*, and *security_change_sid* interfaces. It returns the current process SID or the related object SID if the security server is not yet initialized depending on the security class. Although there are currently no situations where the function is called prior to initialization, it is possible that future development will introduce such cases. If such cases do arise in the future, a better solution would be to preload the security server state with the SIDs that are required to initialize.

If the security server has initialized, then the *unlocked_security_compute_sid* function looks up the security contexts for the SID pair in the SID hash table. The function then sets the user identity for the new context based on which interface is being used, and it initializes the role and type based on the security class. The function then looks for a type rule in the policy configuration. If a type rule exists, then the type is changed accordingly.

The function then applies class-specific logic. For a process, if a transition is being requested, the function checks for a role transition rule and changes the role if a rule is found. If there is no change in the process atributes, then the function simply returns the SID of the process. For an object, if there is no change in the object attributes from the related object, then the function simply returns the SID of the related object.

The function then sets the MLS attributes from the process context. It then calls the *policydb.c:policydb_context_isvalid* function to verify that the security context is valid. If the context is not valid, then the function returns an error. Otherwise, it calls the *sidtab_context_to_sid* function to obtain a SID that corresponds to the context and returns.

3.5.3 sid_to_context

The *unlocked_security_sid_to_context* function panics if it is called before the security server has initialized, unless the SID is predefined. In this case, this function returns a string containing the name of the initial SID. This permits the AVC to call this function for a SID pair when writing an audit record prior to the initialization of the security server.

If the security server has initialized, then this function looks up the security context for the SID in the SID hash table. It then calls the *services.c:context_struct_to_string* function. This function computes the length of the security context string, calling *mls.c:mls_compute_context_len* to obtain the length of the MLS fields of the string if the MLS policy is enabled. It then allocates a buffer of that length using *kmalloc*, copies the user, role, and type names into the buffer, and calls *mls.c:mls_sid_to_context* to write the MLS attributes into the buffer. The function then returns.

3.5.4 context_to_sid

The *unlocked_security_context_to_sid* function panics if it is called before the security server has initialized, unless the context is simply the name of an initial SID. In this case, this function returns the corresponding initial SID. This is not necessary, but it is provided to parallel the *unlocked_security_sid_to_context* function.

If the security server has initialized, then this function creates a copy of the security context string that it can modify as it parses the string. It then looks up the user name, role name, and type name from the string and sets the values in a security context structure for these fields. This function calls *mls.c:mls_context_to_sid* to set the MLS fields in the security context structure based on the remainder of the string. Then, it calls *policydb_context_isvalid* to verify that the context is valid. If the context is valid, the function calls the *sidtab_context_to_sid* function to obtain a SID that corresponds to the context and returns. Otherwise, it returns an error.

3.5.5 load_policy

The *security_load_policy* function calls *policydb_read* to create an in-memory representation of the new configuration. It then applies the *services.c:validate_class* function to each entry in the class hash table to verify that each class that is defined under the existing policy is still defined with the same attributes in the new policy. Since the class and permission values are compiled into the object managers, the security server cannot permit its values for existing classes and permissions to change during system operation.

After checking the classes, the *security_load_policy* function applies the *services.c:convert_context* function to each entry in the SID hash table to convert the values of users, roles, types, sensitivities and categories in the security context structure for each SID to the corresponding values in the new policy. This function calls *mls.c:mls_convert_context* to convert the MLS fields of the structure. After converting all of the values, this function also calls *policydb_context_isvalid* to verify that the context is still valid under the new policy. If it is not, then the SID is removed from the SID hash table.

The *security_load_policy* function then installs the new policy configuration as the active policy, increments the *latest_granting* counter, and calls the *avc_ss_reset* inter-

PERMISSION(S)	DESCRIPTION
compute_av	Compute access vectors
notify_perm	Notify about permissions
transition_sid	Compute new object SID
member_sid	Compute member SID
change_sid	Compute relabel SID
sid_to_context	Obtain context
context_to_sid	Obtain SID
load_policy	Load new policy
get_sids	Get active SIDs

Table 1: Permissions for the security object class.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
compute_av	security	compute_av	current	security
notify_perm	security	notify_perm	current	security
transition_sid	security	transition_sid	current	security
member_sid	security	member_sid	current	security
change_sid	security	change_sid	current	security
sid_to_context	security	sid_to_context	current	sid
context_to_sid	security	context_to_sid	current	security
load_policy	security	load_policy	current	file
get_sids	security	get_sids	current	security

Table 2: Control requirements for security calls.

face of the AVC component to reset the AVC. The global spin lock (*ss_lock*) is released before calling the AVC. This is necessary because the AVC invokes any callback functions registered by the object managers for resets, and these callback functions may perform permission checks to revalidate permissions that are retained in the state of the object managers.

3.5.6 Other interfaces Since none of the implemented policies base their decisions on the history of completed operations, the *security_notify_perm* function simply returns immediately when called. This function is currently never called since the kernel object managers have not yet been changed to notify the AVC of operation completion.

The *security_fs_sid*, *security_port_sid*, *security_netif_sid*, *security_node_sid*, and *security_nfs_sid* functions look for a matching entry from the policy configuration. If no entry is found, then these functions return the appropriate initial SIDs. If an entry is found, then these functions check to see if a SID has already been allocated for each security context in the entry. If not, then these functions call *sidtab_context_to_sid* to obtain a SID for each security context in the entry and cache the SID in the entry. These functions then return the cached SIDs.

3.5.7 System Call Controls The security server prototype defines a *security* class with a set of permissions to control the ability of applications to use the security server system calls, as shown in Table 1. The predefined *security* initial SID is used as the target SID for most of these permission checks. The *load_policy* permission check uses the SID of the configuration file as the target SID to permit control over the files used for policy configurations. The *sid_to_context* permission check uses the SID parameter as the target SID to permit individual control over access to security contexts. The permissions currently required to invoke each system call are shown in Table 2. These permission checks are implemented in the system call functions in *syscalls.c*.

The *context_to_sid* permission check could be changed to similarly use the SID associated with the context parameter as the target SID. However, this is not currently useful, since the SID has already been allocated at that point. If SID descriptors are implemented, then this check should be changed to use the SID descriptor. In that case, the SID descriptor can be released if the check fails.

4 Access Vector Cache

The access vector cache (AVC) is a new operating system component that provides caching of access decision computations to minimize the performance overhead of the Flask security mechanisms. This section describes the set of interfaces provided by the AVC to the kernel object managers. It then describes the set of interfaces provided by the AVC to the security server. Finally, this section describes the implementation of the AVC.

4.1 Interfaces for the Kernel

The data types and function prototypes for the AVC interfaces provided for the kernel object managers are in the *include/linux/flask/avc.h* header file. These interfaces are used by the kernel object managers to perform permission checks and to notify the AVC of completed operations. This subsection describes each of the data types and interfaces used by the kernel object managers. For each data type and function prototype, the type or prototype definition is listed followed by a description of the type or function.

```
void avc_init(void);
```

The *avc_init* function initializes the AVC. The kernel calls this function after the support for dynamic memory allocation has been initialized (*init/main.c:start_kernel*) so that the AVC may allocate memory using *kmalloc*. Alternatively, the AVC could be changed to reserve low memory for its use during the kernel initialization.

```
typedef struct avc_entry_ref {
        avc_entry_t *ae;
} avc_entry_ref_t;

#define AVC_ENTRY_REF_INIT(h) \
{ (h)->ae = NULL; }

#define AVC_ENTRY_REF_CPY(dst,src) \
(dst)->ae = (src)->ae
```

The AVC entry reference type (*avc_entry_ref_t*) consists of a pointer to an entry in the AVC. The AVC returns a reference to the entry used for a permission check. An object manager may save this reference with the corresponding object for subsequent use in other permission checks on the object. An object manager must initialize a reference before its first use with the *AVC_ENTRY_REF_INIT* macro. An object manager may copy a reference with the *AVC_ENTRY_REF_CPY* macro. AVC entry references should only be dereferenced by the AVC functions.

```
typedef struct avc_audit_data {
    char    type;
#define AVC_AUDIT_DATA_FS   1
#define AVC_AUDIT_DATA_NET  2
    union     {
        struct {
            struct dentry *dentry;
            struct inode *inode;
        } fs;
        struct {
            char *netif;
            struct sk_buff *skb;
            struct sock *sk;
            __u16 port;
            __u32 daddr;
        } net;
    } u;
} avc_audit_data_t;

#define AVC_AUDIT_DATA_INIT(_d,_t) \
{ memset((_d), 0, \
  sizeof(struct avc_audit_data)); \
  (_d)->type = AVC_AUDIT_DATA_##_t; }
```

The AVC audit data type (*avc_audit_data_t*) consists of object or parameter information provided by the object manager for the AVC to use when a permission check is audited. This data supplements the audit information directly available to the AVC (*i.e.* the SID pair, the class, the requested permissions, and information about the current process). The *type* field indicates what type of data is being provided by the object manager to the AVC. Currently, two types are supported: file system (*AVC_AUDIT_DATA_FS*) and networking (*AVC_AUDIT_DATA_NET*). The *AVC_AUDIT_DATA_INIT* macro may be used to initialize the data with a specified type.

If the file system type is used, then the object manager may set either of the fields in the *fs* structure to identify the file involved in a permission check. If a *dentry* for the file is available, then the *dentry* field should be set. Otherwise, the *inode* for the file may be set.

If the networking type is used, then the object manager may set any of the fields in the *net* structure. The *netif* field may be set to identify a network interface. The *skb* field may be set to identify a packet. The *sk* field may be set to identify a socket. The *port* field may be set to identify a port number. The *daddr* field may be set to identify an IPv4 address.

```
inline int avc_has_perm_ref_audit(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t requested,
        avc_entry_ref_t *aeref,
        avc_audit_data_t *auditdata);
```

The *avc_has_perm_ref_audit* inline function determines whether the *requested* permissions are granted for the specified SID pair and class. If *aeref* refers to a valid AVC entry for this permission check, then the referenced entry is used. Otherwise, this function obtains a valid entry and sets *aeref* to refer to this entry. To obtain a valid entry, this function first searches the cache. If this fails, then this function calls the *security_compute_av* interface of the security server to compute the access vectors and adds a new entry to the cache. If the appropriate audit access vector (*auditallow* or *auditdeny*) in the entry indicates that the permission check should be audited, then this function audits the permission check, using the *auditdata* parameter to supplement the audit information.

This function returns `0` if permission is granted. If the security server returns an error upon a *security_compute_av* call, then this function returns that error. If the security server returns a sequence number that is less than the latest policy change sequence number, then this function discards the security server response and returns `-EAGAIN`. If permission is denied, then this function returns `-EACCES`.

The kernel object managers call this function to perform permission checks. Kernel object managers may also use variants of this function, such as *avc_has_perm*, *avc_has_perm_audit*, and *avc_has_perm_ref*, in order to omit the reference or audit data parameters. Kernel object managers may also use macro versions of this function, such as *AVC_HAS_PERM_REF*, *AVC_HAS_PERM*, and *AVC_HAS_PERM_AUDIT*, in order to automatically include the class name in the permission symbol.

```
inline int avc_notify_perm_ref(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t requested,
        avc_entry_ref_t *aeref)
```

The *avc_notify_perm_ref* inline function notifies the AVC component that an operation associated with the *requested* permissions has completed successfully. If any of the *requested* permissions are in the *notify* access vector of the corresponding AVC entry, then this function calls the *security_notify_perm* interface of the security server to notify the security server that the operation has completed successfully. If *aeref* refers to a valid AVC entry for the *requested* permissions, then the referenced entry is used to obtain the *notify* vector. Otherwise, this function obtains a valid entry and sets *aeref* to refer to this entry in the same manner as *avc_has_perm_ref_audit*.

This function returns `0` if the notification is successful. If the security server returns an error upon a *security_compute_av* call or a *security_notify_perm* call, then this function returns that error. If the security server returns a sequence number that is less than the latest policy change sequence number, then this function discards the security server response and returns `-EAGAIN`.

The kernel object managers have not yet been changed to call this function. Kernel object managers may also use a variant of this function, *avc_notify_perm*, in order to omit the reference parameter. Kernel object managers may also use macro versions of this function, such as *AVC_NOTIFY_PERM_REF* and *AVC_NOTIFY_PERM*, in order to automatically include the class name in the permission symbol.

```
#define AVC_CALLBACK_GRANT  1
#define AVC_CALLBACK_TRY_REVOKE 2
#define AVC_CALLBACK_REVOKE     4
#define AVC_CALLBACK_RESET      8
#ifdef CONFIG_FLASK_AUDIT
#define AVC_CALLBACK_AUDITALLOW_ENABLE  16
#define AVC_CALLBACK_AUDITALLOW_DISABLE 32
#define AVC_CALLBACK_AUDITDENY_ENABLE   64
#define AVC_CALLBACK_AUDITDENY_DISABLE 128
#endif
#ifdef CONFIG_FLASK_NOTIFY
#define AVC_CALLBACK_NOTIFY_ENABLE     256
#define AVC_CALLBACK_NOTIFY_DISABLE    512
#endif

int avc_add_callback(
    int (*callback)(
            __u32 event,
            security_id_t ssid,
            security_id_t tsid,
            security_class_t tclass,
            access_vector_t perms,
            access_vector_t *out_retained),
    __u32 events,
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
```

```
    access_vector_t perms);
```

The *avc_add_callback* function registers an object manager callback function *callback* with the AVC component for policy change notifications. When the security server calls an AVC interface that corresponds to an event in the set *events* with a SID pair, class and permissions that match *ssid*, *tsid*, *tclass* and *perms*, the AVC component calls the registered *callback* function with the parameters provided by the security server. The *callback* function may then update any affected permissions that are retained in the state of the object manager. The wildcard SID, *SECSID_WILD*, may be used for the *ssid* and *tsid* parameters to match all SID values. Permission vectors match if they have a non-null intersection. The meaning of each event value is explained in the description of the corresponding interface in the next subsection. Callback functions have not yet been implemented for the kernel object managers, so this function is not currently called.

4.2 Interfaces for the Security Server

The function prototypes for the AVC interfaces provided for the security server are in the *include/linux/flask/avc_ss.h* header file. These interfaces are used by the security server to manage the cache as needed for policy changes. This subsection describes each of these interfaces. For each interface, the function prototype is listed followed by a description of the interface.

```
int avc_ss_grant(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno);
```

The *avc_ss_grant* function grants previously denied permissions for a SID pair and class. The wildcard SID, *SECSID_WILD*, may be used for the *ssid* and *tsid* parameters to match all SID values. This function adds the permissions in *perms* to the *allowed* vector in any matching entries in the cache. It then calls any callbacks registered by an object manager for the *AVC_CALLBACK_GRANT* event with a matching SID pair, class and permissions. Permission vectors match if they have a non-null intersection. This function updates the latest policy change sequence number to the greater of its current value and the *seqno* value.

```
int avc_ss_try_revoke(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno,
        access_vector_t *out_retained);
```

The *avc_ss_try_revoke* function tries to revoke previously granted permissions for a SID pair and class, but only if they are not retained in the state of an object manager. If any of the permissions in *perms* are retained, the retained permissions are returned in *out_retained*. The wildcard SID, *SECSID_WILD*, may be used for the *ssid* and *tsid* parameters to match all SID values. This function calls any callbacks registered by an object manager for the *AVC_CALLBACK_TRY_REVOKE* event with a matching SID pair, class and permissions. Permission vectors match if they have a non-null intersection. Each callback is expected to identify which matching permissions are retained in the state of the object manager. The set of retained permissions returned by each callback is added to *out_retained*. This function then removes any permissions in *perms* that were not retained from the *allowed* vector in any matching entries in the cache. This function updates the latest policy change sequence number to the greater of its current value and the *seqno* value.

```
int avc_ss_revoke(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno);
```

The *avc_ss_revoke* function revokes previously granted permissions for a SID pair and class, even if they are retained in the state of an object manager. The wildcard SID, *SECSID_WILD*, may be used for the *ssid* and *tsid* parameters to match all SID values. This function removes any permissions in *perms* from the *allowed* vector in any matching entries in the cache. It then calls any callbacks registered by an object manager for the *AVC_CALLBACK_REVOKE* event with a matching SID pair, class and permissions. Permission vectors match if they have a non-null intersection. Each callback is expected to revoke any matching permissions that are retained in the state of the object manager. This function updates the latest policy change sequence number to the greater of its current value and the *seqno* value.

```
int avc_ss_reset(__u32 seqno);
```

The *avc_ss_reset* function flushes the cache and revalidates all permissions retained in the state of the object managers. This function invalidates all entries in the cache. It then calls any callbacks registered by an object manager for the *AVC_CALLBACK_RESET* event. Each callback is expected to revalidate permissions that are retained in the state of the object manager by calling *avc_has_perm_ref_audit* or one of its variants. This function updates the latest policy change sequence number to the greater of its current value and the *seqno* value.

```
int avc_ss_set_auditallow(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno,
        __u32 enable);
```

The *avc_ss_set_auditallow* function enables or disables auditing of granted permissions for a SID pair and class. The wildcard SID, *SECSID_WILD*, may be used for the *ssid* and *tsid* parameters to match all SID values. The *enable* flag should be `1` to enable auditing and `0` to disable auditing. This function adds or removes, depending on the value of *enable*, the permissions in *perms* from the *auditallow* vector in any matching entries in the cache. It then calls any callbacks registered by an object manager for the *AVC_CALLBACK_AUDITALLOW_ENABLE* or *AVC_CALLBACK_AUDITALLOW_DISABLE* event with a matching SID pair, class and permissions. Permission vectors match if they have a non-null intersection. This function updates the latest policy change sequence number to the greater of its current value and the *seqno* value.

```
int avc_ss_set_auditdeny(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno,
        __u32 enable);
```

The *avc_ss_set_auditdeny* function enables or disables auditing of denied permissions for a SID pair and class. It has the same behavior as *avc_ss_set_auditallow*, except that it modifies the *auditdeny* vector and it is associated with the *AVC_CALLBACK_AUDITDENY_ENABLE* and *AVC_CALLBACK_AUDITDENY_DISABLE* events.

```
int avc_ss_set_notify(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t perms,
        __u32 seqno,
        __u32 enable);
```

The *avc_ss_set_notify* function enables or disables notification of completed operations for a SID pair and class. It has the same behavior as *avc_ss_set_auditallow*, except that it modifies the *notify* vector and it is associated with the *AVC_CALLBACK_NOTIFY_ENABLE* and *AVC_CALLBACK_NOTIFY_DISABLE* events.

4.3 Implementation

This subsection describes the implementation of the AVC. The *include/linux/flask/avc.h* header file contains the inline AVC functions that are called by the kernel object managers. The *kernel/avc.c* source file contains the rest of the implementation of the AVC.

The *avc_init* interface is implemented in *avc.c*. This function allocates memory for all of the cache entries using *kmalloc* and adds them to an internal free list. This function also allocates a page of memory using *__get_free_page* to use as a buffer for the *fs/dcache.c:d_path* function when creating pathnames for audit records.

Both *avc_has_perm_ref_audit* and *avc_notify_perm_ref* are implemented as inline functions in *avc.h*. Each of these two functions disables interrupts locally and takes a single global spin lock (*avc_lock*) on entry using *spin_lock_irqsave*, and each function uses *spin_unlock_irqrestore* before returning. The *avc_lock* is released before calling *security_compute_av* and reacquired upon the return from that call so that the AVC is not locked during the access vector computation by the security server. Similarly, the *avc_lock* is released before calling *security_notify_perm*. The two inline AVC functions call *avc.c:avc_lookup* to search the cache and *avc.c:avc_insert* to add a new entry to the cache. The *avc_has_perm_ref_audit* function calls *avc.c:avc_audit* to audit permission checks.

The *avc.c:avc_audit* function uses *printk* to log whether permission was granted or denied, the names of the requested permissions, the security contexts of the SID pair, and the name of the class. If the current process has a nonzero PID, then the PID and executable path of the current process are also logged. The executable path is determined by using the *fs/dcache.c:d_path* function on the executable's *dentry*.

If file system audit data is set, then the path, device number and inode number are logged. The path is likewise determined using *d_path*. If networking audit data is set, then information about each field that is set in the *net* structure is logged. If the socket field is set to an *AF_INET* socket, then the local and foreign addresses of the socket are logged. If the socket field is set to an *AF_UNIX* socket, then the path or abstract name is logged. If the abstract name space was used, then the initial NULL character is replaced with an "@" character. If the packet field is set to an *IPv4* packet, then the source and destination addresses are logged. If the network interface field is set, then the name is logged.

The *avc_ss_grant*, *avc_ss_try_revoke*, *avc_ss_revoke*, *avc_ss_set_auditallow*, *avc_ss_set_auditdeny* and *avc_ss_set_notify* functions are implemented in *avc.c*. Each of these functions calls *avc.c:avc_control* with the corresponding event value. The *avc_control* function calls *avc.c:avc_update_cache* to update any matching entries in the cache, and *avc_control* calls each of the callbacks registered for the event with matching parameters. If the event is *AVC_CALLBACK_TRY_REVOKE*, then *avc_update_cache* is not called until after the callbacks have been called, since the function must obtain the set of retained permissions from the callbacks. The *avc_update_cache* function disables interrupts locally and takes the global spin lock (*avc_lock*) on entry, releasing the lock and enabling interrupts before returning. The *avc_control* function disables interrupts and takes the lock only to update the latest policy change sequence number.

The *avc_ss_reset* function is also implemented in *avc.c*. This function disables interrupts and takes the global spin lock to invalidate the cache. Then, after enabling interrupts and releasing the lock, this function calls each of the callbacks registered for the *AVC_CALLBACK_RESET* event. Finally, the *avc_ss_reset* function disables interrupts and takes the lock again to update the latest policy change sequence number.

5 Process Management

This section describes the design and implementation of the Flask security mechanisms for Linux process management.

5.1 Design

In this section, the design of the Flask security mechanisms in the Linux process management component is described. This section begins by discussing the object classes that require control. Next follows a discussion of the new permissions. Then the control requirements for the process management system calls are outlined. Finally, the changes to the API are described.

5.1.1 Object Classes Processes are the major abstraction in the process management component. The *process* object class was defined for this abstraction. When a process is created, it is assigned the SID of its parent. That SID may only be changed when a new program is executed. Unless otherwise specified, the new SID depends on the old SID and the SID of the new program. Since the computation of the new security context may involve policy-specific logic, it must be computed by the security server.

An additional object class, *capability*, was defined to control the use of Linux capabilities. It is sufficient to only check capability use, but it could also be useful to place controls over their distribution that could augment the current approach . However, at this time that will not be done.

PERMISSION(S)	DESCRIPTION
execute	Execute
transition	SID Transition
entrypoint	Enter via program
sigkill sigstop sigchld signal	Signal
fork	Fork
ptrace	Trace
getsched	Get schedule info
setsched	Set schedule info
getsession	Get session
getpgid	Get process group
setpgid	Set process group
getcap	Get capabilities
setcap	Set capabilities

Table 3: Permissions for the process object class.

5.1.2 Permissions Table 3 shows the permissions defined for the process management component. The process *execute* permission is used to control the ability of a process to execute from a given executable image. This is distinct from the file *execute* permission which is used to control the ability of a process to initiate the execution of a program. Since Linux on the x86 does not support read without execute permissions on memory pages, the best degree of control that can be obtained in secure Linux will be from the process *execute* permission check that is done. The implication of this is that it will be possible for a process to be tricked into executing memory that was written as data. This problem would not be present on other architectures that do not have this limitation.

The *transition* permission is used to control the ability of a process to transition from one SID to another. The *entrypoint* permission is used to control what programs may be used as the entry point for a given process SID. This permission is similar to the process *execute* permission, except that it is only checked when a process transitions to a new SID. Hence, the security policy can distinguish between what programs may be used to initially enter a given process SID and the full set of programs that may be executed by that process SID.

This *entrypoint* permission is especially necessary in an environment with shared libraries, since most processes must be authorized to execute the system dynamic loader. Without separate control over entry point programs, any security label could be entered by executing the system dynamic loader. Separate entry point control is also necessary in order to support security label transitions on scripts, since the new security label must be authorized to execute the interpreter and the script.

Separate permissions for each signal could easily be defined, but until empirical evidence suggests this is necessary, this will not be done. Separate permissions were defined for the *SIGKILL* and *SIGSTOP* signals, *sigkill*, *sigstop* respectively, since these signals cannot be caught or ignored. A separate permission, *sigchld* was also defined to control the *SIGCHLD* signal because experience demonstrated that it was useful to control this signal separately. A single permission, *signal*, will be used to control the remaining signals.

The *ptrace* permission is used to control the ability of a process to trace another process. The *getsched*, *setsched*, *getsession*, *getpgid*, *setpgid*, *getcap*, and *setcap* permissions are used to control the ability of a process to observe or modify the corresponding attributes of another process. Additional potential controls for scheduling, sessions, process groups, and capabilities are discussed in Section 10.

Currently, a separate permission for each Linux capability is defined for the capability object class. This allows control over all of the abstractions for which capabilities are currently defined. In the future, the control points for each capability will require reexamination to determine if the capability permission is sufficient to control the resource.

5.1.3 Control Requirements Table 4 shows the control requirements for process management system calls. In it, the control requirements for each system call are specified, where each control requirement is described by the class, permission, source SID, and target SID used in a permission check. Since multiple calls may have the same requirements, more than one call may be listed in the leftmost column of a single table entry. In this case, all of the requirements in that table entry apply to all of the calls. In the table, the *path* target SID indicates that the permission check should be applied to each directory in the path prefix, and the directory class is abbreviated by *dir*.

CALL(S)	CONTROL REQUIREMENT(S) Class	Permission	Source SID	Target SID
execve	Dir	search	Current	Path
	File	execute	Current	File
	Process	transition	Current	New
	Process	entrypoint	New	File
	Process	execute	New	File
	Process	ptrace	Parent	New
	FD	inherit	New	FD
kill	Process	sigkill sigstop sigchld signal	Current	Target
wait	Process	sigkill sigstop sigchld signal	Child	Current
fork clone	Process	fork	Current	Current
uselib	Process	execute	Current	File
ptrace	Process	ptrace	Current	Target
getpriority getscheduler getparam	Process	getsched	Current	Target
setpriority setscheduler setparam	Process	setsched	Current	Target
getsid	Process	getsession	Current	Target
getpgid	Process	getpgid	Current	Target
setpgid	Process	setpgid	Current	Target
capget	Process	getcap	Current	Target
capset	Process	setcap	Current	Target

Table 4: Process Management Control Requirements.

The *execve* system call is the most complicated process management call to control. There are two relevant file system permission checks, the *search* check between the process SID and SID of each component of the pathname to verify that the program can be accessed and the *execute* check between the current process SID and new program's SID to verify that the program can be initiated by a process with that SID. The file system controls are described in Section 6.1. A process *execute* check between the new process SID and the new program SID is done to verify that the new process image can execute in the security context of the process.

Since Linux supports a variety of binary formats that must be handled during the *execve* system call, not only must access to the program that is to be executed be controlled, but access to whatever programs are used to support the execution of that program must also be controlled. An example of this is that when a script is executed, access to the script must be checked as well as access to the interpreter of the script. Similarly, *execute* checks for shared libraries are needed. In addition, execution checks will be placed in the Linux-specific system call, *uselib*, to control a process' ability to specify a particular shared library during execution.

The ability of a process to change its SID must be carefully controlled. This is done during the *execve* processing since this is the only place where a process' SID may change. Whether the new SID is specified or results from a default transition, the *transition* permission is checked between the old and new SIDs, and the *entrypoint* permission is checked between the new SID and the program SID. SID transitions on executable scripts are not prevented as is currently done in Linux with setuid transitions. Transitions are prevented, however, if the process is sharing parts of the process state, such as the file descriptor table or signal handlers, as could be the case when certain values for the flags are supplied to *clone*. If the process is being traced, then the *ptrace* permission is checked between the parent process and the new SID.

When a SID transition does occur it is also necessary to revalidate any descriptors. The need to control file descriptors is further discussed in section 6.1. The *inherit* permission for file descriptors is checked for each open descriptor. Any descriptor that does not pass the check will be closed. One consequence of this is that it is quite possible that *stdin*, *stdout*, and *stderr* could be unexpectedly closed on an *execve*. This is only an inconvenience except in the construction of command pipelines. Several options to minimize this impact exist. It may be possible to address this solely within the current framework with correct policy specification. It may prove worthwhile to control these descriptors separately from the rest. It may also be practical to modify the shell or construct special wrapper programs to handle descriptor inheritance and security transitions properly. The issue is still being studied.

The *sigkill*, *sigstop*, *sigchld* and *signal* permissions were added to control whether particular signals may be sent to a process with a given SID. As signals are only generated from within the kernel or local processes, permission checking will only be done when the signal is sent and will not be required when it is received. Before a signal can be delivered the appropriate permission is checked between the sender and receiver SIDs. Because the *fcntl* call can be used to set the recipient of a signals generated from asynchronous I/O, the SID of the process must be saved in the description to allow appropriate signal checking to be done when the kernel generates the signal.

The ability of one process to wait on another needs to be controlled because information can be passed with the exit status. Originally, the design called for a process *wait* permission. It was planned that whenever a security context transition was to occur, this permission would be checked to determine if the parent process would be able to wait on the child. If so, then normal processing could proceed. However, if the parent was to be forbidden to wait on the child, the child would have been reparented to the *init* process and the parent awakened with an appropriate error status. This approach had to be abandoned because it proved difficult to guarantee the process group semantics of Linux.

The same effect can be more cleanly achieved with the signal permissions. When the *wait* system call is executed, the process will only be allowed to wait if there is a child process that matches the argument to the call that is permitted to send its exit signal to the process. This exit signal is set during process creation and can not be changed. If the permission check fails and no other matching children processes that can send their exit signal to the parent exist, the calling process is given an error message indicating that no child was found.

When a process undergoes a SID transition, it is possible that the policy will no longer permit a signal to be delivered to any processes waiting on that process when the transformed process terminates. To ensure that a waiting process is not left waiting in such situations, SID transitions cause waiting processes to be awakened. Waiting will continue only if it is in accordance with the policy.

Control of the *exit* system call is not required. The two issues associated with it, receiving exit status information and being signaled by a child process, are handled by the checking done for the *wait* call and signal mechanism. A side effect of this decision is that a zombie process may be retained in the process table until its parent dies since its parent may be prevented from reaping it. In that case, the zombied process will have to be reparented to the *init* process for reaping, a mechanism already present in Linux. To make this design work properly, it must be possible for all processes, regardless of their security domain, to signal the *init* process to ensure that it will be able to reap orphaned processes. This could be guaranteed using the security policy mechanism or through code modifications to the signal mechanism.

An additional ramification of using signal controls to handle wait and exit notifications comes as a result of a POSIX requirement (POSIX 3.2.2.2) to signal any process group with stopped jobs which becomes orphaned as a result of an exit. If a process has undergone a SID transition after it has done one or more forks, its death will cause a signal to be sent to those children even though the policy might prohibit it.

Linux presents an additional issue with regard to signals and exiting. It is possible for a process to set the signal that it will receive when its parent exits. The checking in the signal mechanism will determine if this signal can be delivered, but it may be desirable to control the ability to use the *prctl* system call which sets this signal. This issue is still being explored.

Because SID transitions do not occur during the *fork* system call, most security policies would not require the explicit control of this call. Some policies, however, may have a need to restrict a process' ability to create a new process. The *fork* permission was added to support such policies. It is checked during calls to *fork* and its Linux-specific generalization *clone*.

The *ptrace* call is controlled using the *ptrace* permission. This permission is initially checked upon a *PTRACE_ATTACH* or *PTRACE_TRACEME* request. In the case of *PTRACE_TRACEME*, the permission is checked between the parent process and the calling process. Otherwise, it is checked between the calling process and the target process. The permission is also revalidated on the other *ptrace* requests since the calling process may have changed its SID or the policy may have changed. As described earlier, the *ptrace* permission is also checked during *execve* if the SID of a traced process changes. Finally, the *ptrace* permission is checked when a process attempts to access the *mem* file of another process in the *procfs* file system.

The *getpriority*, *sched_getscheduler*, and *sched_getparam* calls are controlled using the *getsched* permission. The *setpriority*, *sched_setscheduler*, and *sched_setparam* calls are controlled using the *setsched* permission. The *getsid*, *getpgid*, and *setpgid* calls are controlled using the *getsession*, *getpgid*, and *setpgid* permissions, respectively. The *capget* and *capset* calls are controlled using the *getcap* and *setcap* permissions, respectively. These permissions are checked between the calling process and the target process if they differ.

Most system calls that require superuser privileges to run should also be controlled by the policy. For these calls, it may only be necessary to assign a permission that determines if a process with a given SID can execute the call. Since the Linux capability mechanism already controls many of these calls, the capability permissions are used to make them subject to the central security policy. The source and target SIDS used in the capability permission checking are both that of the current process. Capabilities are discussed further in Section 10.

System calls that only permit a process to observe its own private state or to modify its own unprivileged private state typically do not require controls. Some of these calls are listed in Table 5. Other process management system calls may need to be controlled by the policy. A review of the system call interface to determine the set of calls that need additional controls is described in Section 10.

Editorial Note:

Tables 4 and 5 respectively will be augmented to include all PM system calls that do or do not require control.

CALL(S)	DESCRIPTION
get*uid get*gid getgroups getitimer getpgrp getpid getppid getrlimit getrusage	Obtain current process information
signal sigaction sigalstack sigprocmask sigpending sigsuspend	Signal handling
nanosleep pause	Pause execution

Table 5: Process Management System Calls without Control Requirements.

`execve_secure(..., sid)`
: Execute a file with a specified SID.

`getsecsid()`
: Get the SID of current process.

`getosecsid()`
: Get the SID of current process prior to the last *execve*.

Figure 15: New Linux process management system calls for security-aware applications.

5.1.4 API extensions

Figure 15 lists the new process management system calls for security-aware applications. A new call, *execve_secure*, was added to allow a security-aware application to specify a new SID for the transformed process resulting from the execution of a new program. Currently, this the only way to allow a process to specify a SID to which it will transition. The *execve* call will be a wrapper around this call that requests the transition SID to be calculated by the security policy. Two other system calls, *getsecsid* and *getosecsid*, were added to allow a process to get its SID and its SID prior to the last *execve* call respectively.

STRUCT	FIELD
task	sid osid avc_ref
linux_binprm	sid
fown_struct	sid

Table 6: Changes to process management data structures for labeling.

5.2 Implementation

In this section, the implementation of the Flask security mechanisms in the Linux process management component is described. This section begins by discussing the implementation of support for labeling process management objects. Then, the implementation of the new system calls is described. Finally, the mapping of the control requirements to the code is specified.

5.2.1 Labeling Only minimal modifications to Linux data structures are required to support the process management labeling requirements, as shown in table 6. New fields for the SID of a process and its SID prior to the last call to *exec* were added to the *task* structure. To allow the system to function properly, the *INIT_TASK*, defined in *include/linux/sched.h* had to be modified to initialize these new fields to the initial SID defined in *flask/initial_sids*. A pointer into the access vector cache, *avc_ref*, was also added to the *task* structure to be used as a performance enhancing hint to the access vector cache entry likely to contain the results of permission checking for that process. A SID field was required in the *linux_binprm* structure which is used during exec processing to prepare the transformed binary image of the process. Lastly, a SID field was also required in the fown_struct to allow proper permission checking on signals generated by asynchronous I/O.

5.2.2 API Extensions The existing Linux API was extended to include an *execve_secure* system call which has one additional parameter to specify the security context for the transformed process. The main routine for *execve* processing, *do_execve* in *fs/exec.c*, was renamed to be *do_execve_secure*, and an additional parameter was added for the SID of the specified context. A new *do_execve* that calls *do_execve_secure* with a null SID was added to handle the existing *execve* call. When a null SID is encountered during processing, the security server is consulted via the *security_transition_sid* interface for a default SID that will be used.

The two new process management system calls were straightforwardly implemented in the *sys_getsecsid* and *sys_getosecsid* routines added to *kernel/sched.c*. Like other similar calls, these take no arguments and return the appropriate element of the *task* structure as pointed to by *current*. As the information is requested only in the context of the calling process, no security checking was required for these calls.

All three calls were added to the new security library *libsecure* and a new header file, *proc_secure.h* was created. Additionally, new *_secure* versions of the other forms of *exec* which allow the specification of a security context were added to this library. Alternatively, these could have, and probably should have, been placed in the C library. They were added to this new library for ease of implementation and for portability reasons and will likely be moved to the C library in the future.

5.2.3 Control Requirements The process management permissions were defined in *flask/access_vectors* and interpreted for each component policy in the appropriate files in *policy*. It is through these policy files that the need to allow every process to signal the init process was addressed. Permission checks using the AVC interface were added at various places throughout the kernel as needed.

A convenient location to place the *execve* access checks was in the *prepare_binprm* kernel routine used in the implementation of the *execve* call. This routine was the natural choice because it is used for loading the executable requested in the system call arguments and also any other executables indicated by the binary image header. The specified SID for the new process image was made accessible from the *linux_binprm* structure built for the call. The other SID values necessary for permission checking were already accessible from within this routine. In general, placing the access checks in this routine made it unnecessary to place additional checks in all of the individual binary handlers. However, it was necessary to add *process_execute* and *file_execute* checks in the ELF binary handler since it was possible that it could call other interpreters that otherwise would have gone unchecked. These checks were added to *do_load_elf_binary* in *fs/binfmt_elf.c*.

Since shared libraries are loaded using *mmap*, a process *execute* check was needed in the *old_mmap* routine defined in *arch/i386/kernel/sys_i386.c*. Section 6.1.3 describes the complete control requirements of *mmap*. This check, however, is not sufficient on the x86 architecture since a file may be mmapped read only and still be executed. This is actually an instance of the general problem of not being able to control execution of anything that a process can read. The security impact of this particular problem and the best way to minimize it are still being investigated.

Whenever a call to *execve* is going to result in a change in security context, additional action must be taken to ensure that the policy can not be violated. The call is aborted with the global variable *errno* set to EPERM when there is inappropriate sharing that resulted from a previous call to *clone*. Similarly, the call is aborted with EPERM if the process is being traced and the parent process lacks *ptrace* permission to the new SID for the process. Open descriptors must be revalidated with the *inherit* permission and closed when necessary. This is done in a new function, *revalidate_fds* which is modeled after the *flush_old_files* routine used to check the *close_on_exec* flag, called from the *flush_old_exec* routine. Finally, the *compute_creds* routine was modified to update the *sid* and *osid* fields of the task structure and to call *wake_up_interruptible* on the parent to force permission checking if the parent was waiting on the transformed process. If the process is not waiting, this action is harmless. A small side effect to this approach is that it is possible for a parent process to notice that its child has undergone a SID transition which prevents it from waiting.

Linux currently checks if signals may be delivered in the *send_sig_info* routine defined in *kernel/signal.c* which is the central control point for the signal mechanism. The appropriate signal permission checks were placed immediately after the existing checks. When the checks fail, the global variable *errno* is set to EACCES. Linux's signal checking for signals resulting from asynchronous I/O is done in the *send_sigio* routine defined in *fs/fcntl.c*. Here too, the permission checking is done following the existing checks. On failure, no signal is sent.

Signal checking is also used to control a process' ability to wait on another. Checks to determine if a child's *exit_signal* can be delivered to the parent were added to the *sys_wait4* routine in *kernel/exit.c*. Whenever a process is awakened, Linux checks to see if the *wait* call should return or if the process should be placed back to sleep. At this time, the permission checks are repeated to ensure that the waiting process can continue to wait. If not, the *wait* call returns with the global variable *errno* set to ECHILD. This ensures that the waiting process will not be blocked indefinitely. In this case, when the child eventually exits, it will remain a zombied process until it can be reaped by the *init* process.

The execute permission check for a shared library specified in *uselib* was placed in *sys_uselib*. Failure aborts the call with the global variable *errno* set to EACCES.

No special changes to *fork* or *clone* were necessary to handle the initialization of the new fields of the *task* structure. When the *init* process is properly initialized during system startup, those fields are inherited from the parent process during process creation automatically without modification to the existing code. Since *fork* is implemented as a special case of *clone*, only the *clone* call actually needed modification. The permission checking was added to the *do_fork* routine defined in *kernel/fork.c*. The SID of the current process was used twice in the call to the AVC. Failure aborts the call with the global variable *errno* set to EACCES.

The *ptrace* permission check was added to the *ptrace* system call (*arch/i386/kernel/ptrace.c:sys_ptrace*), the *execve* call (*fs/exec.c:must_not_trace_exec_flask*), and the access routines for the *mem* file in *procfs* (*fs/proc/mem.c:get_task*). The scheduling, session, and process group permission checks were added to the corresponding system calls in *kernel/sched.c* and *kernel/sys.c*. The *setcap* and *getcap* permission checks were added to the corresponding system calls in *kernel/capability.c*.

The checking for all of the capability permissions was centralized in a single location, the *capable* function defined in *include/linux/sched.h*. Because the capability checks are done here, not all of the context information that might make more interesting security policies possi-

OBJECT CLASS
pipe
directory
regular file
symbolic link
character device
block device
FIFO
socket file
file system
file description

Table 7: Object classes for the Linux file system component.

ble is available. This limits the check to only the current process SID and prevents the ability to limit the use of a capability on a per-object basis as was possible in the DTOS system. The AVC reference in the task structure was used for these permission checks. An important note is that the capability permissions correspond to the capability definitions in *linux/include/linux/capability.h*. The implementation of the checking mechanism is dependent on the correct ordering of the permission definitions with respect to the capability definitions.

6 File System

This section describes the design and implementation of the Flask security mechanisms for the Linux file system.

6.1 Design

This section describes our design for integrating the Flask security mechanisms into the Linux file system component. It begins with a discussion of the object classes and permissions defined for the file system component. This is followed by a description of the control requirements for the system calls used to manage and perform directory and file operations. Then, the approach for providing persistent labels for files, directories, and file systems is discussed. Finally, the new file-related system calls defined for security-aware applications are described.

6.1.1 Object Classes The logical abstractions provided by the Linux file system component were studied to determine the set of object classes that needed to be labeled and controlled by the Flask mechanisms. The set of object classes for the file system component is shown in Table 7. Three abstractions that need to be controlled, pipes, files, and directories, were immediately evident in the Linux API. Files were further refined into separate object classes for each file type defined by the Linux API, *i.e.* regular files, symbolic links, character devices, block devices, FIFOs, and Unix domain socket files.

When a pipe is created, it inherits the SID of the creating process by default. When a directory or file is created, it is assigned a SID that represents the security context in which it is created by default. This context depends on the security context of the creating process and the security context of the parent directory. Since the computation of the new security context may involve policy-specific logic, it must be computed by the security server.

Although file systems are not treated as first-class objects in the Linux API, a separate object class was defined for the file system abstraction. Entire file systems are labeled not only to control operations such as mounting and unmounting but also to represent the aggregate label of all files within the file system.

Finally, an object class was defined for the *file description* abstraction. The term *file description* is used by POSIX [3] to describe the information referenced by a file descriptor, *e.g.* the file offset, file status and file access modes for an open file. File descriptors may be inherited across *execve* calls, and they may be transferred through IPC. Consequently, it is necessary to label and control file descriptions. The SID of a file description is inherited from the SID of the process that created it.

6.1.2 Permissions For each object class, a set of permissions was defined to control access to objects in that class. These permissions were identified by studying the services provided by the Linux file system component. For each service, the objects whose state is observed or modified by the service were identified, and permissions for the corresponding object classes were defined.

Table 8 shows the permissions defined for controlling access to the pipe object class and to the file object classes. Unlike the existing Linux file permissions, which only control the ability to open a file, the Flask *read* and *write* permissions are defined for the actual ser-

PERMISSION(S)	DESCRIPTION
read	Read
write	Write or append
append	Append
poll	Poll/select
ioctl	IO control
create	Create
execute	Execute
access	Check accessibility
getattr	Get attributes
setattr	Set attributes
unlink	Remove hard link
link	Create hard link
rename	Rename hard link
lock	Lock or unlock
relabelfrom relabelto transition	Relabel

Table 8: Permissions for the pipe and file object classes.

PERMISSION(S)	DESCRIPTION
add_name	Add a name
remove_name	Remove a name
reparent	Change parent directory
search	Search
rmdir	Remove
mounton mountassociate	Use as mount point

Table 9: Additional permissions for the directory object class.

vices of reading from a file or writing to a file. The implications of this stricter definition are discussed further in Section 6.1.3. A separate *append* permission was defined to support append-only access to a file. Whereas the existing Linux access controls permit certain services based only on the attributes of the directory, such as the service for obtaining a file's attributes and the services for adding, removing or renaming a hard link to a file, Flask provides finer-grained control through corresponding file permissions such as *getattr*, *link*, *unlink*, and *rename*. Three permissions are defined for the relabel service, since it is useful to control the relationship between each pairing of the three SIDs involved: the SID of the subject, the old SID of the file, and the new SID for the file.

Table 9 shows the additional permissions defined for

PERMISSION(S)	DESCRIPTION
mount	Mount
remount	Change options
unmount	Unmount
getattr	Get attributes
relabelfrom relabelto transition	Relabel
associate	Associate file

Table 10: Permissions for the file system object class.

PERMISSION(S)	DESCRIPTION
create	Create
getattr	Get attributes
setattr	Set attributes
inherit	Inherit across execve
receive	Receive via IPC

Table 11: Permissions for the file description object class.

manipulating directories. Three separate permissions are provided for adding entries to directories, removing entries from directories and changing the parent entry (the **..** entry) of a directory during a rename. In contrast, the Linux access controls use the *write* access mode for all three services. Two permissions are defined for the mount service, where the *mounton* permission is used to control the ability of a subject to mount on a given mountpoint and the *mountassociate* permission is used to control the relationship between the mounted directory and the mountpoint.

Permissions for controlling access to file systems are shown in Table 10. Permissions are provided for controlling mounting and unmounting and for obtaining file system attributes, such as the number of free blocks. As with files, three permissions are defined for the relabel service. The *associate* permission controls what files are permitted in the file system.

Table 11 lists the permissions for controlling access to file description objects. Each of these permissions is implicitly granted for file description objects with the same SID as the subject. The *getattr* and *setattr* permissions control services that observe or modify the flags and the file offset of the file description. The *inherit* and *receive* permissions control the service of inheriting descriptors across an *execve* and the service of receiving descriptors through IPC, respectively.

6.1.3 Control Requirements After defining permissions for the services provided by the Linux file system component, control requirements were defined for each Linux system call that provides one or more of these services. The control requirements specify the permissions that must be granted for the system call to successfully execute.

In the following tables, the control requirements for each system call are specified, where each control requirement is described by the class, permission, source SID (SSID), and target SID (TSID) used in a permission check. Since multiple calls may have the same requirements, more than one call may be listed in the leftmost column of a single table entry. In this case, all of the requirements in that table entry apply to all of the calls.

In the tables, the *path* target SID indicates that the permission check should be applied to each directory in the path prefix. File system classes and SIDs are abbreviated by *fs*, file description classes and SIDs are abbreviated by *fd*, and directory classes and SIDs are abbreviated by *dir*. A file permission check uses the class of the file being accessed, so the *file* class in the tables may be the pipe class, the directory class, or any of the file object classes.

Several of the system calls listed in the tables have two forms, one of which takes a pathname parameter and the other takes a file descriptor parameter, *e.g. stat* and *fstat*. In the tables, this is expressed as *(f)stat*. The corresponding control requirements are identical except that the descriptor-based call naturally does not have the *search* requirement.

Table 12 shows the control requirements for system calls that manipulate files. The control requirements listed in this table for the *open* system call are the requirements for opening an existing file rather than the requirements for creating a new file. The process must be able to search the directories in the path prefix, and it must be able to create the file description. The *read*, *write* and *append* requirements on the *open* system call are enforced in accordance with the flags to *open*. The *write* permission grants either write access or append access. The *append* permission is only checked if *write* permission is not granted and the *O_APPEND* flag is specified.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
open	dir fd file file file	search create read write append	current current current current current	path fd file file file
read, readv, pread	fd file	setattr read	current current	fd file
write, writev, pwrite	fd file file	setattr write append	current current current	fd file file
sendfile	fd file fd file file	setattr read setattr write append	current current current current current	in_fd in_file out_fd out_file out_file
mmap mprotect	fd file file file process	setattr read write append execute	current current current current current	fd file file file file
(f)stat lstat	dir file	search getattr	current current	path file
(f)chmod, (f)chown, lchown, (f)truncate, utime(s)	dir file	search setattr	current current	path file
access	file	access	current	file
poll, select	file	poll	current	file
fcntl: F_GETLK, F_SETLK, F_SETLKW flock	file	lock	current	file
ioctl: FIBMAP	file	getattr	current	file
ioctl: FIONREAD	fd file	getattr getattr	current current	fd file
ioctl: FIGETBSZ	file	getattr	current	file
ioctl: GETFLAGS, GETVERSION	file	getattr	current	file
ioctl: SETFLAGS, SETVERSION	file	setattr	current	file
ioctl	file	ioctl	current	file

Table 12: Control requirements for manipulating files.

Since the *read*, *write*, and *append* permissions are intended to control the actual services of reading from a file and writing (or appending) to a file, it is necessary to verify that the permissions are still granted when those services are performed. The prior checks during the *open* call may no longer be valid, since the process may have changed SID, the file may have changed SID, a different process may be using the file description, or a change in the security policy may have occurred. Hence, the system calls which implement those services, such as the *read*, *write* and *sendfile* system calls, must revalidate the permissions obtained during *open*. The calls must also verify that *setattr* permission to the file description parameters is granted, since the file offset is modified by these calls.

When a file is mapped into memory via the *mmap* call, the *read*, *write*, and *append* permissions are revalidated. However, the permissions may become invalid while the file is still mapped. Consequently, the permissions must be revalidated when pages are read from the file or written to the file, and the pages for a file in the page cache must be invalidated when the file is relabeled or a policy change that would affect access to the file occurs. The *mmap* call must also check the process *execute* permission to control the ability of a process to execute from a particular shared library. The *mprotect* call must also revalidate these permissions when the current protection is changed.

Table 13 shows the control requirements for system calls that manipulate directories. In addition to requiring *search* permission to directories in the path prefix, the *chdir*, *fchdir*, and *chroot* system calls require *search* permission to the last component of the path. The control requirements listed in this table for the *open* and *creat* system calls are the requirements for creating a new file. The process must have *search* permission to the directories in the path prefix, *create* permission to the file description, *add_name* permission to the parent directory, and *create* permission to the new file. Furthermore, the file must have *associate* permission to the file system. The requirements for *mkdir*, *mknod*, and *symlink* only differ from the requirements for *open* in that there is no file description.

The *rename* system call requires *search* permission to both paths, *remove_name* permission to the old parent

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
(f)chdir,	dir	search	current	path
chroot	dir	search	current	dir
open,	dir	search	current	path
creat	fd	create	current	fd
	dir	add_name	current	parent
	file	create	current	file
	fs	associate	file	fs
mkdir,	dir	search	current	path
mknod,	dir	add_name	current	parent
symlink	file	create	current	file
	fs	associate	file	fs
rename	dir	search	current	oldpath
	dir	remove_name	current	oldparent
	file	rename	current	file
	dir	reparent	current	file
	dir	search	current	newpath
	dir	add_name	current	newparent
	dir	remove_name	current	newparent
	file	unlink	current	newfile
	dir	rmdir	current	newfile
link	dir	search	current	path
	dir	add_name	current	parent
	file	link	current	file
unlink	dir	search	current	path
	dir	remove_name	current	parent
	file	unlink	current	file
rmdir	dir	search	current	path
	dir	remove_name	current	parent
	dir	rmdir	current	dir
getdents,	fd	setattr	current	fd
readdir	dir	read	current	dir
readlink	file	read	current	file

Table 13: Control requirements for manipulating directories.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
remount	dir	search	current	path
	fs	remount	current	fs
mount	dir	search	current	devpath
	dir	search	current	dirpath
	fs	mount	current	fs
	dir	mounton	current	dir
	dir	mountassociate	root	dir
umount	dir	search	current	path
	fs	unmount	current	fs
ustat	fs	getattr	current	fs
(f)statfs	dir	search	current	path
	fs	getattr	current	fs

Table 14: Control requirements for manipulating file systems.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
lseek, llseek	fd	setattr	current	fd
fcntl: F_SETOWN, F_SETSIG	fd	setattr	current	fd
fcntl: F_SETFL	fd file	setattr write	current current	fd file
fcntl: F_GETFL, F_GETOWN, F_GETSIG	fd	getattr	current	fd
ioctl: FIONBIO, FIOASYNC	fd	setattr	current	fd

Table 15: Control requirements for manipulating descriptions.

directory, *rename* permission to the file and *add_name* permission to the new parent directory. If the file being renamed is a directory, and its parent directory would be changed by the rename, then *reparent* permission must be granted to the file. If a file already exists at the new pathname, then *remove_name* permission must be granted to the new parent directory and *unlink* permission or *rmdir* permission must be granted to the existing file or directory.

Table 14 shows the control requirements for system calls that manipulate file systems. The *remount* call in the table represents the *mount* system call used with the *MS_REMOUNT* flag. The *mount* call in the table represents the *mount* system call used to mount a file system. Mounting a file system requires *search* permission to both the device special file pathname and to the mount point pathname, *mount* permission to the file system, and *mounton* permission to the mount point directory. The root directory of the file system must have *mountassociate* permission to the mount point directory.

Table 15 shows the control requirements for system calls that manipulate file descriptions. If a file is opened with the *O_APPEND* flag, and this flag is subsequently cleared via the *F_SETFL* command of the *fcntl* system call, then *write* permission must be granted to the file. The other system calls in this table only observe or modify the state of the file description itself, so they only require *getattr* or *setattr* permission to the file description. The *F_SETOWN* and *F_SETSIG* commands to the *fcntl* system call must also be checked against process management control requirements to ensure that the calling process may cause signals to be sent to the owner.

Note that Table 15 does not include entries for *fcntl.F_SETFD*, *fcntl.F_GETFD*, *ioctl.FIONCLEX* or *ioctl.FIOCLEX*. These operations may be used to observe or modify the close-on-exec flag of a file descriptor. The close-on-exec flag of a file descriptor is private to that file descriptor and is not part of the file description state. Hence, this flag is not shared and access to it does not require any permissions.

6.1.4 Persistent Labeling Since file systems, files, and directories are persistent objects, an approach for providing persistent labels for these objects was developed. To ensure that the security attributes of these objects are preserved even if the file system is moved to another system, the Linux file system component must maintain a table within each file system that specifies the security context of the file system and each file and directory within that file system. This approach also ensures that the security attributes are preserved over time, even if the policy changes, and that the security attributes can be interpreted by a user if a manual translation of attributes for a new policy is required.

The Linux file system component can handle security contexts without sacrificing policy flexibility or performance by treating security contexts as opaque strings and by mapping these labels to SIDs by a query to the security server for internal use by the file system com-

`open_secure(..., fd_sid, f_sid)`
Open a file with a file description labeled *fd_sid*. If creating, create the new file with label *f_sid*.

`mkdir_secure(..., sid)`
Create a directory labeled *sid*.

`mknod_secure(..., sid)`
Create a node labeled *sid*.

`symlink_secure(..., sid)`
Create a symbolic link labeled *sid*.

`stat_secure(..., sidp)`
Get file SID of *pathname*.

`lstat_secure(..., sidp)`
Get symbolic link SID of *pathname*.

`fstat_secure(..., sidp)`
Same as above, except using a fd.

`statfs_secure(..., sidp)`
Get filesystem SID of file system for *pathname*.

`fstatfs_secure(..., sidp)`
Same as above, except using a fd.

`chsid(pathname, sid)`
Relabel file *pathname* to *sid*.

`fchsid(fd, sid)`
Same as above, except using a fd.

`lchsid(pathname, sid)`
Relabel symbolic link *pathname* to *sid*.

`chsidfs(pathname, fs_sid, f_sid)`
Relabel the filesystem for *pathname*.

`fchsidfs(fd, fs_sid, f_sid)`
Same as above, except using a fd.

Figure 16: New Linux file-related system calls for security-aware applications.

ponent. For efficient storage, the file system component may assign a fixed-size value, referred to as a *persistent SID* (PSID), to each security context associated with an object in the file system, and may then partition the persistent labeling table into a mapping between each PSID and its security context and a mapping between each object and its PSID. The PSID is purely an internal abstraction within the file system and has a distinct name space for each file system. Hence, PSIDs may be lightweight and the allocation of PSIDs may be optimized for each file system.

6.1.5 API extensions To permit applications to create objects with a specified label rather than the default label, an extended form of each of the file creation system calls must be added that accepts an additional SID parameter. To permit applications to obtain the SID of an object, an extended form of each of the file status system calls must be added that return an additional SID parameter. To permit applications to change the SID of an object, new system calls must be added. The new Linux system calls that must be added for security-aware applications are shown in Figure 16.

For the new system calls that are simply extended forms of existing Linux system calls, the same set of control requirements apply. The control requirements for the new system calls for relabeling are shown in Table 16.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
(f)chsid	dir	search	current	path
	file	relabelfrom	current	file
	file	relabelto	current	new
	file	transition	old	new
	fs	associate	new	fs
(f)chsidfs	dir	search	current	path
	fs	relabelfrom	current	fs
	fs	relabelto	current	new_fs
	fs	transition	old	new_fs
	fs	associate	new_file	new_fs

Table 16: Control requirements for relabeling.

6.2 Implementation

In this section, our implementation of the Flask file security mechanisms in the Linux file system component is described. The implementation began by adding support for labeling file system objects, followed by the addition of the new system calls. Finally, the control requirements were implemented.

6.2.1 Labeling The kernel data structures were studied to identify the structures used internally for mounted file systems (*struct super_block*), active files and directories (*struct inode*), and file descriptions (*struct file*). All three of these structures are defined in *include/linux/fs.h*. Since these structures are private to the kernel and have no specific size requirements, a SID field was added to each structure. Since a *struct inode* object is used to represent all types of files, an object class field was also added to the structure.

Two other private kernel data structures also required the addition of SID fields. The inode attributes structure (*struct iattr*) is used for changing the attributes of a file, so a SID field was added to this structure, and a corresponding flag (*ATTR_SID*) was defined to indicate that the SID is being changed. The file description owner structure (*struct fown_struct*) is used to store the identity of the process that set the owner on a file description, so a SID field was added to this structure.

The implementation of the persistent labeling table was partitioned into a component that is independent of the file system type (*fs/psid.c*) and a set of components that are specific to each file system type. Most of the implementation resides in the filesystem-independent component; hence, persistent labeling support for additional file system types may be easily added. The filesystem-independent component implements the mapping between each PSID and its security context using regular files in a fixed subdirectory of the root directory of each file system. Two PSIDs are reserved: PSID 0 represents the default label to assign to unlabeled objects in the file system, and PSID 1 represents the label of the file system itself. The subdirectory and its files are always treated as being labeled with a fixed security context so that the security policy may control access to the mapping. Synchronous writes are used to update the mapping files and the writes are ordered to ensure that there are no dangling references.

The interface to the filesystem-independent component is defined in *include/linux/flask/psid.h*. The file system calls the *fs/psid.c:psid_init* function to initialize the mapping between PSIDs and security contexts when the file system is mounted (*fs/super.c:mount_root*, *fs/super.c:do_mount*). If the file system is unlabeled, then this function obtains the SIDs for the unlabeled file system from the security server. If the file system is unlabeled and it is being mounted read-write, then this function creates a new PSID mapping on the file system. If an unlabeled file system is mounted read-only initially and is subsequently remounted read-write, then the *fs/psid.c:psid_remount* function creates a new PSID mapping on it when it is called by *fs/ext2/super.c:ext2_remount*. The file system calls the *fs/psid.c:psid_release* function to free any memory and release any files used for the PSID mapping when the file system is unmounted (*fs/super.c:do_umount*).

The filesystem-specific components implement the mapping between each file and its PSID. Currently, the filesystem-specific component has only been implemented for the native Linux file system type, *ext2*. The *ext2*-specific component stores the PSID for each file in a formerly unused field of the on-disk inode structure (*struct ext2_inode*). Since the PSID is readily available in the on-disk inode, no extra overhead is incurred either to obtain the PSID when a file is accessed or to set the PSID when a file is created.

The *ext2fs* code calls the *fs/psid.c:psid_to_sid* function to obtain the SID of an existing inode based on its PSID when the inode is read from the disk (*fs/ext2/inode.c:ext2_read_inode*). The *fs/psid.c:sid_to_psid* function is called to obtain a PSID for an inode based on its SID when an inode is allocated (*fs/ext2/ialloc.c:ext2_new_inode*) or when the SID of an inode is changed (*fs/ext2/inode.c:ext2_notify_change*).

6.2.2 API extensions

Due to the large number of new file-related calls and the potential need for future expansion, the new file-related calls were implemented as library functions that invoke a single system call, *flask_filecall*, with a first parameter that identifies the specific call. Internally, the Linux file system component uses a variant of the Virtual File System (VFS) interface. Extended forms of the file creation operations were added to this interface to permit the filesystem-independent code to pass the SID of the new file to the filesystem-specific code. New operations would not have been necessary if the existing file creation operations accepted a general attribute structure as a parameter, as in the BSD VFS interface.

SID parameters needed to be added to several internal functions to support the new system calls. Some of these internal functions are called from many different locations within the kernel and may be called from kernel-loaded modules. Consequently, it was not practical to simply change the existing function and update all calls to it. For such functions, *_secure* was appended to the function name, and the interface and implementation of the function were extended for the new processing. A stub function that merely calls the new function with default parameters was added using the old function name and interface. This permits the existing code to continue to use the old function interface, but introduces the overhead of an extra function call in these cases.

6.2.3 Control Requirements The Linux file system code was studied to determine the location to implement each permission check associated with the control requirements specified in the design. Several factors influenced the placement of permission checks. Whenever possible, the permission checks were implemented in the filesystem-independent code so that they are applied to all file system types. Only a few control requirements are specific to the *ext2* file system type, so this was feasible for almost all of the permission checks. Permissions are typically checked as early as possible in the processing of each system call to simplify cleanup from permission failures and to ease maintenance of the checks as the file system code evolves. For services that are also controlled by a Linux access control check, the Flask permission check was usually implemented at the same location. However, a Flask permission cannot be checked until the kernel data structures for the necessary objects are accessible and appropriately locked. This required deferring some of the Flask permission checks until a later point in the processing.

To reduce the overhead of permission checks, the file system component was changed to store references to AVC entries with its file description (*struct file*) objects and inode (*struct inode*) objects. Two reference fields were added to the *struct file*: one for the permissions granted to the file description, and one for the permissions granted to the file. Since many of the file operations use pathname parameters rather than file descriptors, a field for storing a reference to the AVC entry containing the permissions granted to the file was also added to the *struct inode*.

Table 17 shows the control requirements implemented in each kernel function used to manipulate files. Only the class and permission are shown for each control requirement; the source SID and target SID can be found in the corresponding design table. The functions that implement the services for reading and writing files revalidate the permissions initially checked during *open_namei*, using the AVC entry reference in the file description. Since these permissions also migrate into

FUNCTION(S)	CONTROL REQUIREMENT(S)	
	CLASS	PERM
filp_open	fd	create
open_namei	file file file	read write append
sys_read, sys_pread, filemap_nopage	fd file	setattr read
do_readv_writev	fd file file file	setattr read write append
sys_write, sys_pwrite, filemap_write_page	fd file file	setattr write append
sys_sendfile	fd file fd file file	setattr read setattr write append
old_mmap sys_mprotect	fd file file file process	setattr read write append execute
cp_new_stat, cp_old_stat	file	getattr
inode_change_ok (SID)	file file file fs	relabelfrom relabelto transition associate
inode_change_ok (other)	file	setattr
sys_access	file	access
do_select, do_poll	file	poll
sys_fcntl: F_GETLK, F_SETLK, F_SETLKW sys_flock	file	lock
file_ioctl: FIBMAP	file	getattr
file_ioctl: FIONREAD	fd file	getattr getattr
file_ioctl: FIGETBSZ	file	getattr
ext2_ioctl: GETFLAGS, GETVERSION	file	getattr
ext2_ioctl: SETFLAGS, SETVERSION	file	setattr
sys_ioctl	file	ioctl

Table 17: Implementing the file control requirements.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
lookup_dentry	dir	search
sys_(f)chdir, sys_chroot	dir	search
may_create	dir dir	search add_name
open_namei, do_mkdir, do_mknod, do_symlink	file fs	create associate
do_link	file	link
may_delete	dir dir	search remove_name
vfs_unlink	file	unlink
vfs_rmdir	dir	rmdir
vfs_rename	file	rename
vfs_rename_dir	dir dir	reparent rmdir
vfs_rename_other	file	unlink
sys_getdents, old_readdir	fd dir	setattr read
sys_readlink	file	read

Table 18: Implementing the directory control requirements.

the page cache for memory-mapped files, any cached pages from a file are invalidated when the SID of a file is changed. As a result, subsequent access to the pages will cause *filemap_nopage* to be executed. The *inode_change_ok* function implements both the permission checks for changing the SID of a file and the permission check for changing the ordinary Linux attributes of a file. Only the control requirements implemented in *ext2_ioctl* are specific to the filesystem type.

The control requirements implemented in each kernel function used to manipulate directories are shown in Table 18. The *lookup_dentry* function checks *search* permission on each directory in the path prefix. All of the system calls use this function to perform pathname lookup. The functions for changing the current and root directories must also perform a *search* permission check against the last component of the path. The *may_create* and *may_delete* functions provide convenient locations for the Flask control requirements on the containing directory. The *unlink* and *rmdir* permission checks had to be implemented separately from the *may_delete* function, since those checks should not be applied when the *rename* call removes the old link. The *sys_getdents* and *old_readdir* functions revalidate *read* permission using the AVC entry reference in the file description.

The remaining tables (Table 19 and Table 20) show the control requirements implemented in the kernel functions used to manipulate file systems and file descriptions. The *mount* and *mountassociate* permissions depend on the SID of the file system being mounted and the SID of the root directory of that file system. Consequently, these checks are deferred until after the file system metadata has been loaded, just prior to linking the file system into the file system name space.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
do_remount	fs	remount
do_mount	fs dir dir	mount mounton mountassociate
do_umount	fs	unmount
sys_ustat	fs	getattr
sys_(f)statfs	fs	getattr
sys_(f)chsidfs	fs fs fs fs	relabelfrom relabelto transition associate

Table 19: Implementing the file system control requirements.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
sys_lseek, sys_llseek	fd	setattr
sys_fcntl: F_SETOWN, F_SETSIG	fd	setattr
sys_fcntl: F_SETFL	fd file	setattr write
sys_fcntl: F_GETFL, F_GETOWN, F_GETSIG	fd	getattr
sys_ioctl: FIONBIO, FIOASYNC	fd	setattr

Table 20: Implementing the file description control requirements.

7 Other File System Types

This section describes how the Flask file security mechanisms were applied to control access to several additional file system types. The section begins by discussing the analysis, design, and implementation of labeling for the *procfs* file system. It then discusses the design and implementation of labeling for the *devpts* file system. Finally, it discusses the design and implementation of labeling for NFS client support.

7.1 Procfs

This subsection begins with an analysis of the *procfs* file system and its implementation. The design for labeling *procfs* files is then described. Finally, the implementation of labeling and controls is discussed.

7.1.1 Procfs Analysis The Linux *procfs* file system provides an interface to kernel data structures as an alternative to the traditional */dev/kmem* interface. This file system is typically mounted at */proc*. The */proc* file system hierarchy is described in the *proc(5)* manual page and in the *Documentation/proc.txt* file.

The Linux *sysctl* system call provides an interface for reading and writing system parameters. This system call is described in the *sysctl(2)* manual page. The system parameters are arranged in a tree structure, and they are typically also accessible through a parallel directory tree under the */proc/sys* subdirectory. In addition to the previously mentioned documents, the */proc/sys* hierarchy is described in the files in the *Documentation/sysctl* directory. Based on the documentation for *sysctl*, it appears that applications should always use the */proc/sys* interface instead of the system call interface for portability across kernel versions.

There is a subdirectory for each running process under */proc*, named by its process identifier. A process may always use the */proc/self* symbolic link to refer to its own subdirectory. The effective uid and effective gid of the process is used for the user and group ownership of the files and subdirectories within each process-specific subdirectory. Several files in these subdirectories permit any user to read them: the *cmdline*, *maps*, *stat*, *statm*, and *status* files. The remaining files may only be read by the owner. The *mem* file, which provides access to the memory of the process, may only be read and written by the owner. The *procfs* implementation only permits a process to access its own *mem* file or the *mem* file of a child process that is stopped and being traced (*fs/proc/mem.c:get_task*). The Linux 2.2.12 kernel implementation does not support write access to the *mem* files due to a risk of overwriting kernel memory if a process dies in the middle of a write, but future versions of the kernel are likely to support such access.

Most of the files in */proc* outside of the process-specific subdirectories are readable by all users. The most notable exceptions are */proc/kmsg* and */proc/kcore*, which are only readable by the superuser. The */proc/kmsg* file is used by *klogd* as a source of kernel log information as an alternative to the *syslog* system call interface. The */proc/kcore* file provides access to the physical memory of the system in core file format, and can be used by *gdb* to examine the current state of any kernel data structures. The kernel implementation also requires that a process possess the *CAP_SYS_RAWIO* capability to open the */proc/kcore* file (*fs/proc/array.c:open_kcore*).

Only the superuser may write to files in */proc* outside of the process-specific subdirectories. Most files that can be written correspond to system parameters and are located in */proc/sys*. A few files outside of */proc/sys* also permit writing for configuration. For example, */proc/mtrr* may be written to manipulate the memory type range register, as described in *Documentation/mtrr.txt*. Some of the files under */proc/ide*, */proc/scsi*, */proc/bus*, and */proc/parport* may be written for device configuration.

The types and functions provided by the *procfs* file system to the kernel are defined in *include/linux/proc_fs.h*. Entries in the */proc* file system are defined by *struct proc_dir_entry* objects. The *proc_register* function may be used to add an entry under a given parent entry, and the *create_proc_entry* function may be used to create and register a dynamically allocated entry given a name, mode, and parent entry. Functions are also provided for registering entries under certain well-defined subdirectories, such as the *net* or *scsi* subdirectories.

When an entry in */proc* is looked up, an inode is obtained for the entry. The *fs/proc/inode.c:proc_get_inode* function copies the owner and mode attributes from the entry into the inode when it is requested by a lookup. This function also calls the entry's *fill_inode* operation. For process-specific files, this operation is implemented by the *base.c:proc_pid_fill_inode* function, which copies the effective uid and effective gid of the associated process into the inode. The *inode.c:proc_read_inode* function also copies the effective identity attributes into the inode when the inode for a process-specific file is initialized.

The types and functions provided by the *sysctl* call to the kernel are defined in *include/linux/sysctl.h*. A sysctl table is defined by an array of *struct ctl_table* objects. Each object may contain a pointer to an array of child objects. The statically declared *kernel/sysctl.c:root_table* contains the base set of sysctl entries. The *sysctl_init* function calls the *register_proc_table* function to create the corresponding entries under */proc/sys*.

Additional sysctl tables may be added dynamically by using the *register_sysctl_table* function. Unlike *proc_register*, this function does not link the new table into an existing table in the hierarchy. Instead, the new table is added to a linked list of top-level tables. Consequently, dynamically-registered tables must contain dummy entries to provide the path from the root of the hierarchy to the newly registered parameters. The *register_sysctl_table* function also calls the *register_proc_table* function on the newly registered table.

When the *sysctl* system call is called, the *kernel/sysctl.c:parse_table* function looks up the appropriate *struct ctl_table* object, calling the *ctl_perm* function to check that the process has search access to each table in the prefix. When a matching entry is found, the *do_sysctl_strategy* function calls *ctl_perm* to check that the process has the appropriate read and/or write access to the table. The *ctl_perm* function is also called by the *do_rw_proc* function when a sysctl parameter is accessed through */proc/sys*.

7.1.2 Procfs Labeling Design

To enable the security policy to control access to each process-specific subdirectory based on the security attributes of the associated process, each process-specific subdirectory and its files will be labeled with the SID of the associated process. If the security policy needs to be able to distinguish the individual files within each process-specific subdirectory, then a new interface could be added to the security server that would return the SID for each file based on the SID of the associated process. However, it is not evident that the security policy will require such distinctions. This contrasts with the distinctions in file modes among the files in the process-specific subdirectory.

Of particular note in the process-specific subdirectories are the *mem* files, since these files have the potential to provide read and write access to the memory of other processes. However, the existing restrictions on access to *mem* seem adequate if the ability to *ptrace* a child process is controlled by the security policy. Such controls need to be added to the process management component.

Most of the files outside of the process-specific subdirectories have the same fundamental security properties: readable by everyone and writeable only by administrators. Consequently, most of these files may be labeled with a single SID. This labeling scheme may be further refined over time to provide better support for least privilege. It seems desirable to provide support for easily specifying a distinct SID at any point in the */proc* hierarchy and automatically assigning that SID to all files below that point that are not explicitly labeled. This will permit gradual refinement of labeling with minimal changes.

Due to the highly sensitive nature of the *kmsg* and *kcore* files, each of these files will be labeled with a distinct SID to permit fine-grained control over access to each file. Note that the Flask capability permission for *CAP_SYS_RAWIO* will also need to be granted for access to the *kcore* file.

As with the Linux access controls for *sysctl*, the Flask controls should be same whether a parameter is accessed through */proc/sys* or through the *sysctl* system call. Hence, file mandatory access controls will be added to the *sysctl* system call code to parallel the controls that are already enforced when */proc/sys* is accessed. As an initial step toward least privilege, the *kernel*, *vm*, *net*, *fs*, and *dev* subtrees will each be labeled with a distinct SID. Additionally, the */proc/sys/kernel/modprobe* file will have a distinct SID to permit fine-grained control over the ability to change the path executed by the kernel to automatically load kernel modules. The other files and directories under */proc/sys* will be labeled with a SID that distinguishes them from the rest of */proc*.

7.1.3 Procfs Labeling Implementation

The *base.c:proc_pid_fill_inode* function and the *inode.c:proc_read_inode* function in *fs/proc* were changed to copy the SID of the associated process into the inode for the process-specific files. The *inode.c:proc_read_super* function was changed to initialize the SID of the root directory and the file system of the *procfs* file system to the *proc* initial SID. The initial SID was declared in *flask/initial_sids* and defined in the security policy configuration.

A *sid* field was added to the *struct proc_dir_entry* structure in *include/linux/proc_fs.h*. The field was added at the end of the structure to ensure that the statically declared structures could be left unchanged. The *proc_root_kcore*, *proc_root_kmsg*, and *proc_sys_root* definitions in *fs/proc/root.c* were changed to set the SID explicitly to a distinct initial SID value. The initial SIDs were declared in *flask/initial_sids* and defined in the security policy configuration.

The *inode.c:proc_get_inode* function was changed to copy the SID from the *struct proc_dir_entry* structure into the inode if the SID is non-null. This change permits entries to be individually labeled by setting the SID in the structure. If the SID is null, then the inode is left unlabeled by *proc_get_inode*.

The *fs/proc/root.c:proc_lookup* and *fs/proc/fd.c:proc_lookupfd* functions were changed to copy the SID from the parent directory inode if the inode is unlabeled after the call to *proc_get_inode*. These changes cause unlabeled entries to be automatically labeled with the SID of their parent directory.

A *sid* field was added to the *struct ctl_table* structure in *include/linux/sysctl.h*. The field was added at the end of the structure to ensure that the statically declared structures could be left unchanged. The *kernel*, *vm*, *net*, *fs*, and *dev* entries in the *kernel/sysctl.c:root_table* definition were changed to set the SID of each entry to a corresponding initial SID. The *modprobe* entry in the *kern_table* definition was changed to set the SID of the entry to a corresponding initial SID. The initial SIDs were declared in *flask/initial_sids* and defined in the security policy configuration.

The *ctl_table_inherit_sid* function was added to *kernel/sysctl.c*. This function traverses a sysctl table and ensures that all entries are labeled, using inheritance from the parent entry as necessary. The *sysctl_init* function was changed to call this function on the *root_table*.

The *ctl_table_root_sid* function was also added to *kernel/sysctl.c*. This function is used to copy the SIDs from the root sysctl table into the dummy entries in a dynamically registered sysctl table. The *ctl_table_inherit_sid* function may then be used to ensure that all of the entries in the dynamically registered sysctl table are labeled properly. The *register_sysctl_table* function was changed to call the two functions.

The *register_proc_table* function was changed to copy the SID of the *ctl_table* structure into the *proc_dir_entry* structure returned by *create_proc_entry*. This change ensures that the */proc/sys* entries are labeled with the same SID as the corresponding *sysctl* entry.

The *ctl_perm* function was changed to check the Flask directory *search* permission when a table entry is being traversed. This function was also changed to check the Flask file *read* and/or *write* permissions when a table entry is being accessed. These changes ensure that the Flask controls are enforced when the *sysctl* system call is used. Since the *ctl_perm* function is also called by *do_rw_proc*, these checks are also redundantly performed when a sysctl parameter is accessed through */proc/sys*.

7.2 Devpts

The *devpts* file system provides an interface to pseudo terminal (*pty*) devices. It is typically mounted at */dev/pts*. A new pty device file is dynamically created when the */dev/ptmx* pty master multiplex device is opened. At mount time, a user identity, group identity, and mode can be specified for all pty files in the *devpts* file system. Typically, this feature is used to set the group and mode to allow write access by programs that are setgid to the *tty* group. A user identity is typically not specified at mount time. In the absence of the corresponding mount option, the user and/or group identity is inherited from the process that created the pty.

As with the *procfs* file system, an initial SID was defined for the *devpts* file system and its root directory. The SIDs of the file system and root directory are set to this value by the *fs/devpts/inode.c:devpts_read_super* function. The *devpts_statfs* function returns this SID as the SID of the file system. The *devpts_read_inode* function returns this SID as the SID of the root inode.

To permit the security policy to control access to individual ptys, the *devpts_pty_new* function was modified to call the security server's *security_transition_sid* interface to obtain a SID for each new pty file. The SID of the creating process and the SID of the root directory of the *devpts* file system are passed to this interface as inputs. The security server returns a SID derived from these two SIDs. Pty files may need to be subsequently relabeled by programs to reflect changes in the label of the associated process. For example, the `login` program could relabel a pty created by `rlogind` based on the initial security context of the user shell.

7.3 NFS client support

To allow the security policy on an NFS client to control access to file systems mounted from ordinary NFS servers, each NFS file will be labeled based on the NFS server identity. A file system security context and a file security context can be specified for each NFS server in the policy configuration. These contexts are applied to all file systems and all files mounted from the NFS server.

An initial SID is defined as the default SID for NFS file systems and their files. If security contexts are not defined for the NFS server in the policy configuration, then the *security_nfs_sid* function returns this initial SID. Otherwise, the *security_nfs_sid* function returns the SIDs that correspond to the security contexts in the configuration.

The *fs/nfs/inode.c:nfs_read_super* function obtains the SIDs for the file system and root directory from the security server using the *security_nfs_sid* function. The *nfs_statfs* function returns the file system SID. The *nfs_fill_inode* function copies the inode SID from the SID of the root directory. The *nfs_notify_change* function returns `EACCES` if the SID is being changed, or it checks *setattr* permission otherwise.

Separate labels could be supported for different file systems mounted from the same NFS server, but this would require the *nfs_read_super* function to pass an additional parameter to *security_nfs_sid* to identify the particular file system. Since the *mount* call is only provided with the NFS file handle for the root directory (as opposed to the pathname on the server), this is currently not implemented. If the `mount` program were modified to also pass the pathname, then the configuration could specify security contexts based on both the server identity and the pathname on the server for the root directory.

OBJECT CLASS
TCP socket
UDP socket
raw IP socket
Unix stream socket
Unix datagram socket
node
network interface

Table 21: Object classes for the Linux networking component.

8 Networking

This section describes the design and implementation of the Flask security mechanisms for Linux networking.

8.1 Design

This section describes our design for integrating the Flask security mechanisms into the Linux networking component. It begins with a discussion of the object classes and permissions defined for the networking component. This is followed by a description of the control requirements for the system calls used to manage and perform network interprocess communication. Finally, the new socket system calls defined for security-aware applications are described.

8.1.1 Object Classes The object classes for the Linux implementation of the *AF_INET* and *AF_UNIX* protocol families are shown in Table 21. Since Linux uses the BSD socket API, the socket is the principal controlled object class. The socket object class was refined into separate object classes for the different types of sockets. When a socket is created via the *socket* call, it inherits the SID of the process that created it by default. If the socket is created by a connection, then it inherits the SID of the listening socket by default. An alternative approach would be to have the security server compute the SID of the new socket based on the SID of the listening socket and the SID of the client socket.

The Linux network component creates two special purpose sockets for use by the *AF_INET* protocol family. The *tcp_socket* is used to send resets when a TCP packet is rejected, since there may be no local socket corresponding to the packet. The *icmp_socket* is used to send ICMP messages. Two initial SIDs were defined for these sockets, with the corresponding security context determined by the security server.

For socket types that maintain message boundaries, each message is separately labeled. For other socket types, each message is implicitly associated with the SID of its sending socket. Although messages are labeled and controlled, a separate object class is not necessary. When a message is sent on a socket, it inherits the SID of the sending socket by default. When the network component receives a message from the network, the SID of the message is initially set to a default message SID associated with the receiving network interface. This default message SID is computed by the security server. If the message was protected using the IPSEC protocols, then the SID of the received message is set based on the information in the corresponding security association.

Each message is also associated with the SID of its source socket and the desired SID for its destination socket. By default, the desired SID for the destination socket of a message is set to the *any_socket* initial SID. When a message is received from the network, the source socket SID of the message is initially set to the default message SID for the receiving network interface. If the message was protected using IPSEC protocols, then the source socket SID and the destination socket SID are set based on information in the corresponding security association.

The node object class was defined to permit controls on inbound messages based on the source address and to permit controls on outbound messages based on the destination address. The network interface object class was defined to permit controls based on the network interface used to send or receive a message. The SIDs for nodes and the SIDs for network interfaces are computed by the security server.

TCP and UDP port numbers are labeled to permit controls over the ability to bind to particular ports. Only those port numbers which are outside of the range used to automatically bind sockets, *ip_local_port_range*, are labeled and controlled. Like messages, a separate object class is not necessary for port numbers. The security server computes SIDs for the port numbers.

PERMISSION(S)	DESCRIPTION
bind	Bind name
name_bind	Use port or file
connect	Initiate connection
getopt	Get socket options
setopt	Set socket options
shutdown	Shut down connection
recvfrom	Receive from socket
sendto	Send to socket
recv_msg	Receive message
send_msg	Send message

Table 22: Additional permissions for the socket object classes.

If an *AF_UNIX* socket is associated with an object in the file system namespace, there are two different objects with separate SIDs that represent the socket in different ways. The *AF_UNIX* socket object is created first using the *socket* call and it inherits the SID of the creating process by default. The socket file object is created by a subsequent *bind* call on the socket, and it is labeled with a SID computed by the security server based on the SID of the creating process and the SID of the parent directory. The socket file object continues to exist until it is explicitly unlinked from the file system namespace. If an *AF_UNIX* socket is associated with a name in the abstract namespace, there is no separate object for the name.

8.1.2 Permissions

Since sockets are accessed through file descriptions, the socket object classes inherit the permissions defined for controlling access to the file object classes. Only the *read*, *write*, *poll*, *ioctl*, *create*, *lock*, *getattr*, *setattr*, *relabelfrom*, *relabelto*, and *transition* file permissions are meaningful for sockets.

Table 22 shows additional permissions that are specifically defined for controlling access to the socket object classes. The *bind*, *connect*, *getopt*, *setopt*, and *shutdown* permissions control the ability of processes to invoke various socket-specific system calls. For *AF_INET* sockets, the *name_bind* permission controls the relationship between a socket and its port number. For *AF_UNIX* sockets, the *name_bind* permission controls the relationship between a socket and its file. The *recvfrom* and *sendto* permissions control the relationship between the sending socket and the receiving socket for datagrams. The *recv_msg* and *send_msg* permissions control the relation-

PERMISSION(S)	DESCRIPTION
listen	Listen for connections
accept	Accept a connection
newconn	Create new socket for connection
connectto	Connect to server socket
acceptfrom	Accept connection from client socket

Table 23: Additional permissions for the TCP and Unix stream socket object classes.

PERMISSION(S)	DESCRIPTION
getattr	Get attributes
setattr	Set attributes
tcp_recv	Receive TCP packet
tcp_send	Send TCP packet
udp_recv	Receive UDP packet
udp_send	Send UDP packet
rawip_recv	Receive Raw IP packet
rawip_send	Send Raw IP packet

Table 24: Permissions for the network interface object class.

PERMISSION(S)	DESCRIPTION
tcp_recv	Receive TCP packet
tcp_send	Send TCP packet
udp_recv	Receive UDP packet
udp_send	Send UDP packet
rawip_recv	Receive Raw IP packet
rawip_send	Send Raw IP packet
enforce_dest	Enforce destination socket

Table 25: Permissions for the node object class.

PERMISSION(S)	DESCRIPTION
route_control	Manipulate routing tables
arp_control	Manipulate ARP table
rarp_control	Manipulate RARP table
net_io_control	Use device-specific ioctl

Table 26: Additional permissions for the system object class.

ship between a datagram message and the receiving or sending socket. These two permissions are implicitly granted if the message SID is equal to the sending socket SID.

The connection-oriented service provided by stream sockets requires several additional permissions, as shown in Table 23. The *listen* and *accept* permissions control the ability of processes to invoke the corresponding system calls. The *newconn* permission controls the relationship between the server socket created by a connection and the listening socket. This permission is implicitly granted if the sockets have the same SID. The *connectto* and *acceptfrom* permissions control the relationship between the client socket and the server socket.

The set of permissions for the network interface object class is shown in Table 24. The *setattr* and *getattr* permissions control the ability of processes to manipulate the interface parameters. The remaining permissions control the relationship between a message and the network interface on which it is sent or received. Similar permissions are defined for the node object class, as shown in Table 25, to control the relationship between an inbound message and its source address and the relationship between an outbound message and its destination address. The *enforce_dest* permission for the node object class was defined to support the extended socket calls, as described in Section 8.1.4.

Table 26 shows permissions that were added to the system object class for the networking component. The *route_control* permission controls the ability of a process to manipulate the kernel IP routing table. The *arp_control* and *rarp_control* permissions control the ability of a process to manipulate the kernel ARP cache and RARP table, respectively. The *net_io_control* permission controls the ability of a process to invoke a device-specific *ioctl* on a network device.

8.1.3 Control Requirements This subsection describes the control requirements for each Linux system call that provides a service implemented by the network component. The control requirements specify the permissions that must be granted for the system call to successfully execute. In the following tables, the control requirements for each system call are specified, where each control requirement is described by the class, permission, source SID (SSID), and target SID (TSID) used in a permission check. Since multiple calls may have the same requirements, more than one call may be listed in the leftmost column of a single table entry. In this case, all of the requirements in that table entry apply to all of the calls.

CALL(S)	CONTROL REQUIREMENT(S)			
	CLASS	PERM	SSID	TSID
listen	socket	listen	current	so
	socket	newconn	so	newconn_sid
connect	socket	connect	current	client_so
	socket	connectto	client_so	server_so
	netif	tcp_send	client_so	netif
	node	tcp_send	client_so	node
	netif	tcp_recv	server_so	netif
	node	tcp_recv	server_so	node
accept	socket	accept	current	listen_so
	socket	newconn	listen_so	server_so
	socket	acceptfrom	server_so	client_so
	netif	tcp_send	server_so	netif
	node	tcp_send	server_so	node
	netif	tcp_recv	client_so	netif
	node	tcp_recv	client_so	node
write, send, sendto, sendmsg	socket	write	current	so
	socket	connectto	so	peer_so
	socket	acceptfrom	so	peer_so
	netif	tcp_send	so	netif
	node	tcp_send	so	node
	netif	tcp_recv	peer_so	netif
	node	tcp_recv	peer_so	node
read, recv, recvfrom, recvmsg	socket	read	current	so
	socket	connectto	so	peer_so
	socket	acceptfrom	so	peer_so
	netif	tcp_send	so	netif
	node	tcp_send	so	node
	netif	tcp_recv	peer_so	netif
	node	tcp_recv	peer_so	node
	fd	receive	current	fd

Table 27: Control requirements for connection-oriented communication. The *tcp_send* and *tcp_recv* permission requirements only apply to TCP traffic, not Unix stream traffic. The *receive* permission requirement only applies to Unix stream traffic containing file descriptors.

In the tables, network interface classes and SIDs are abbreviated by *netif*. A socket permission check uses the class of the socket being accessed, so the *socket* class in the tables may be any appropriate socket object class. Socket SIDs are abbreviated by *so*. Since a single call may involve multiple sockets, socket SIDs may be prefixed with a distinguishing identifier, such as *listen_*, *client_*, *server_*, or *dst_*.

Table 27 shows the control requirements for system calls used to perform connection-oriented communication. For each of these calls, permission is required to invoke the call on the socket, *i.e.* the *listen*, *connect*, *accept*, *write*, and *read* permissions. For Unix stream sockets that are bound in the file system namespace, the client process must be granted *search* permission to the directories in the path and *write* permission to the socket file in order to use *connect*.

On the server node, the *newconn* permission must be granted between the listening socket and the newly created server socket, and the *acceptfrom* permission must be granted between the server and client sockets. For TCP, these permissions are checked when the server receives the client's *SYN* packet on a listening socket. On the client node, the *connectto* permission must be granted between the client socket and the server socket. For TCP, this permission is checked when the client obtains the label of the server socket from the server's *SYN-ACK* packet. If a TCP simultaneous open occurs, then both nodes check *connectto* permission when they receive the other node's *SYN* packet. The appropriate connection permission (*connectto* or *acceptfrom*) must be revalidated when traffic is sent or received on an established connection, since a policy change may revoke permission for the connection. In this case, the connection must be reset. For TCP, on each node, the appropriate *tcp_send* and *tcp_recv* permissions must be granted for the network interface and the peer node.

For Unix stream sockets, the *connectto* permission check is redundant with the *acceptfrom* permission check, since the connection is local. Nonetheless, both permission checks are performed to maintain consistency with the TCP controls. The network control requirements for Unix stream sockets only differ from the TCP requirements in that there is no equivalent for the *tcp_send* and *tcp_recv* permission checks.

There is also an additional file control requirement for Unix stream or datagram communication, the *receive* requirement on file descriptions. As explained in the file system control design, open file descriptions must be labeled and controlled. The *receive* permission must be granted between the receiving process and each open file description received through Unix stream or datagram communication.

The control requirements for connectionless communication are shown in Table 28. As with the connection-oriented calls, permission is required to invoke each call on the socket, *i.e.* the *connect*, *write*, and *read* permissions. For Unix datagram sockets that are bound in the

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
connect	socket	connect	current	so
write,	socket	write	current	so
send,	socket	sendto	so	dst_so
sendto,	socket	send_msg	so	msg
sendmsg	netif	udp_send	msg	netif
	node	udp_send	msg	node
	netif	rawip_send	msg	netif
	node	rawip_send	msg	node
read,	socket	read	current	so
recv,	socket	recvfrom	so	src_so
recvfrom,	socket	recv_msg	so	msg
recvmsg	netif	udp_recv	msg	netif
	node	udp_recv	msg	node
	netif	rawip_recv	msg	netif
	node	rawip_recv	msg	node
	fd	receive	current	fd

Table 28: Control requirements for connectionless communication. The *udp_send* and *udp_recv* permission requirements only apply to UDP traffic. The *rawip_send* and *rawip_recv* permission requirements apply to any IPv4 traffic other than TCP or UDP. The *receive* permission requirement only applies to Unix datagram traffic containing file descriptors.

file system namespace, the client process must be granted *search* permission to the directories in the path and *write* permission to the socket file in order to use *connect*, *sendto*, or *sendmsg*.

On the sending node, the *sendto* permission must be granted between the source and destination sockets, and the *send_msg* permission must be granted between the source socket and the message. For *AF_INET* sockets, by default, the *any_socket* initial SID is used as the destination socket SID in the *sendto* permission check, since the sending node does not know the SID of the destination socket. On the receiving node, the *recvfrom* permission must be granted between the destination and source sockets, and the *recv_msg* permission must be granted between the destination socket and the message. For IPv4 traffic, on each node, the appropriate *udp_send* and *udp_recv* permissions, or the *rawip_send* and *rawip_recv* permissions, must be granted for the network interface and the peer node.

For Unix datagram communication, the SID of destination socket is known when the *sendto* permission check is performed, so it is used in the check. The *sendto* permission check is redundant with the *recvfrom* permission check, since the communication is local and since the actual destination socket SID is used in the *sendto* permission check. Nonetheless, both permission checks are performed to maintain consistency with the UDP and raw IP controls. The network control requirements for Unix datagram communication only differ from the UDP requirements in that there is no equivalent for the *udp_send* and *udp_recv* permission checks. As with Unix stream communication, there is the additional *receive(fd)* file control requirement when receiving file descriptors.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
socket	socket	create	current	so
bind	socket	bind	current	so
	socket	name_bind	so	port
getsockname	socket	getattr	current	so
getpeername	socket	getattr	current	so
getsockopt	socket	getopt	current	so
setsockopt	socket	setopt	current	so
shutdown	socket	shutdown	current	so

Table 29: Control requirements for other socket calls.

The control requirements for the other socket calls are shown in Table 29. The *create* permission must be granted in order to create a socket with the *socket* call. The other calls all require permission to invoke the call on an existing socket, *e.g.* the *getattr* permission. For *AF_INET* sockets, if the *bind* call is invoked with a port number outside of the range used to automatically bind sockets, then the *name_bind* permission must be granted between the socket and the port number. For *AF_UNIX* sockets, if the *bind* call is invoked with a name in the file system namespace, then the *name_bind* permission must be granted between the socket and the socket file.

The control requirements for the *ioctl* commands are shown in Table 30. The commands for manipulating the attributes of a network interface are controlled through the *setattr* and *getattr* permissions on each network interface. The remaining commands are controlled through system permissions.

8.1.4 API extensions Figure 17 shows the new Linux socket system calls that must be added for security-aware applications. The *getsockname_secure*, *getpeername_secure*, *accept_secure*, *recvfrom_secure*,

CALL(S)	CONTROL REQUIREMENT(S) CLASS	PERM	SSID	TSID
ioctl: TIOC-OUTQ, INQ	socket	getattr	current	so
ioctl: SIOC-GIFADDR, GIFBRDADDR, GIFDSTADDR, GIFNETMASK	netif	getattr	current	netif
ioctl: SIOC-SIFFLAGS, SIFADDR, SIFBRDADDR, SIFDSTADDR, SIFNETMASK	netif	setattr	current	netif
ioctl: SIOC-ADDRT, DELRT, RTMSG	system	route-_control	current	kernel
ioctl: SIOC-DARP, GARP, CSARP	system	arp-_control	current	kernel
ioctl: SIOC-DRARP, GRARP, CSRARP	system	rarp-_control	current	kernel
ioctl: device-specific	system	net_io-_control	current	kernel

Table 30: Control requirements for ioctl commands.

and *recvmsg_secure* calls permit applications to obtain the SIDs of local and peer sockets and the SIDs of messages. The *socket_secure* and *listen_secure* calls permit applications to specify a particular SID to use when a new socket is created. The *listen_secure* call also permits applications to specify that server sockets created by a connection should be labeled with the SID of the client socket. The *sendto_secure* and *sendmsg_secure* calls permit applications to specify a particular SID to use for a message.

The *connect_secure*, *sendto_secure*, and *sendmsg_secure* calls also permit applications to specify a desired SID for the peer socket. For connection requests and outbound datagrams, this restriction can only be enforced by the destination node. However, a destination node may not be capable of enforcing the restriction or it may not be trusted to enforce the restriction. Consequently, the source node performs an *enforce_dest* permission check between the desired SID and the destination node SID. This check is not necessary for *AF_UNIX* sockets, since the communication is local.

When used with stream sockets, *connect_secure* specifies the desired SID of the listening socket. The SID of the server socket created by the connection may differ from this SID since the server application may have used *listen_secure*. If a client wishes to ensure that the server socket has a particular SID prior to sending data, then it may obtain the SID using *getpeername_secure*. Alternatively, a client may specify the desired server socket SID with *sendto_secure*. In this case, since the server socket SID was obtained by the client node during connection establishment, the client node may check the desired SID against it.

Since sockets are accessed through file descriptions, the *fstat_secure* call may also be used to obtain the SID of a socket. The *fchsid* call may be used to relabel UDP sockets, raw IP sockets, Unix datagram sockets or Unix stream sockets. Relabeling of TCP sockets is not supported in the current design because there is no mechanism for synchronizing the change with the peer transport layer. The change might also need to be synchronized with the peer application, because the peer application may be relying on the socket SID provided by the extended socket calls. No mechanism is provided for such

`getsockname_secure(..., sidp)`
Get the address and SID of the local socket.

`getpeername_secure(..., sidp)`
Get the address and SID of the peer socket.

`accept_secure(..., sidp)`
Accept a connection and return the SID of the peer socket.

`recvfrom_secure(..., sso_sidp,msg_sidp)`
Receive a message, its source address, the SID of the source socket, and the SID of the message.

`recvmsg_secure(..., sso_sidp,msg_sidp)`
Same as above, except using the *recvmsg* interface.

`socket_secure(..., sid)`
Create a socket with a specified SID.

`listen_secure(..., sid, useclient)`
Set the state of a socket to accept connections and specify the SID to use for server sockets created by connections. If *sid* is non-zero, then each server socket created by a connection will be labeled with the specified SID. If *useclient* is non-zero, then each server socket created by a connection will be labeled with the SID of its peer socket. It is an error to specify both *sid* and *useclient*.

`connect_secure(..., sid)`
Specify the address and the desired SID of the peer socket. If the socket is of type SOCK_DGRAM, then datagrams may only be sent to or received from a socket with the specified SID. If the socket is of type SOCK_STREAM, then the connection will fail unless the listening socket has the specified SID.

`sendto_secure(..., dso_sid, msg_sid)`
Send a message and specify the desired SID for the destination socket and/or the SID of the message.

`sendmsg_secure(..., dso_sid, msg_sid)`
Same as above, except using the *sendmsg* interface.

Figure 17: New Linux socket system calls for security-aware applications.

STRUCT	FIELD
sock	sclass sid newconn_sid useclient peer_sid
open_request	conn_request_sid newconn_sid
sk_buff	sso_sid dso_sid msg_sid
device	sid default_msg_sid

Table 31: Changes to network data structures for labeling.

synchronization with the peer application for either TCP sockets or Unix stream sockets.

8.2 Implementation

In this section, the implementation of the Flask security mechanisms in the Linux networking component is described. This section begins by discussing the implementation of support for labeling network objects. Then, the implementation of the new socket system calls is described. Finally, the mapping of the control requirements to the code is specified.

8.2.1 Labeling The kernel data structures were studied to identify the structures used internally for sockets (*struct sock* and *struct socket*), open connection requests (*struct open_request*), messages (*struct sk_buff*), and network interfaces (*struct device*). Since these structures are private to the kernel and have no specific size requirements, they were extended to include additional fields, as shown in Table 31.

The *struct sock* structure was extended to include the security class (*sclass*) and the SID (*sid*) of the socket, the SID to use for new sockets created by connections to the socket (*newconn_sid*), a flag to indicate the use of the client SID for this purpose (*useclient*), and the SID of the peer socket (*peer_sid*). The allocator for *struct sock* objects, *sk_alloc*, initializes the security class field to the general socket class, the SID field to the SID of the current process, and the peer SID field to the *any_socket* initial SID. The *inet_create* function sets the security class field to be one of TCP socket, UDP socket, or raw IP socket based on the specified socket type. The *unix_create* function sets the security class field to be either Unix stream socket or Unix datagram socket. The *inet_listen* and *unix_listen* functions set the *newconn_sid* field to the SID of the socket by default. The *udp_connect* and *unix_dgram_connect* functions reset the *peer_sid* field to the *any_socket* initial SID if the association is broken.

The more abstract *struct socket* structure is embedded in an inode structure (*struct inode*), which has a security class and SID field used by the file controls. The allocator for *struct socket* objects, *sock_alloc*, initializes the security class and the SID of the inode to the general socket class and the SID of the current process, respectively. The *inet_create* and *unix_create* functions set the security class field in the inode to the same value as in the *struct sock* object.

The *struct open_request* structure was extended to include the SID of the connection request (*conn_request_sid*) and the SID to use when the socket for the connection is created (*newconn_sid*). This structure temporarily stores these SID values for TCP until the new server socket is created at the completion of the connection establishment.

The *struct sk_buff* structure was extended to include the SID of the source socket (*sso_sid*), the desired SID of the destination socket (*dso_sid*) and the SID of the message (*msg_sid*). The allocator for *struct sk_buff* objects, *alloc_skb*, initializes the source socket SID and the message SID to the *unlabeled* initial SID, and it initializes the destination socket SID to the *any_socket* initial SID. The *skb_clone*, *skb_copy*, and *skb_realloc_headroom* functions preserve the values of these three SID fields when messages are copied. The *ip_defrag* and *ip_glue* functions ensure that all fragments of a message have the same values for the three SID fields and that the three SID fields are set correctly for the complete message.

When a message is allocated from a socket's send buffer, the *sock_wmalloc* function sets the source socket SID and message SID to the SID of the socket, and the destination socket SID to the peer SID of the socket. When an unlabeled message is associated with a sending socket, the *skb_set_owner_w* inline function sets the three SID fields in the same manner. There are two special cases for setting the SID fields of an outbound TCP message. When a *SYN-ACK* is created for a normal connection, *tcp_make_synack* sets the source socket SID and the message SID to the value of the *newconn_sid* field of the *struct open_request* object, so that the *SYN-ACK* is labeled with the SID of the server socket that will be created by the connection rather than the SID of the listening socket. When an *ACK* is sent to complete a connection handshake, the *tcp_send_ack* function sets the destination socket SID to the *any_socket* initial SID, since the listening socket may have a different SID than the server socket.

The *struct device* structure was extended to include the SID of the network interface (*sid*) and the default message SID for the interface (*default_msg_sid*). The *devinet_ioctl* function sets the SID field and the default message SID field of the network interface if it has not been previously set. These SID values are obtained from the security server based on the name of the network interface. When an unlabeled message is received on a network interface, the *ip_rcv* function sets the source socket SID and the message SID to the default message SID of the network interface, and the destination socket SID to the *any_socket* initial SID.

When a TCP *SYN* is received on a listening TCP socket, the *tcp_v4_conn_request* function sets the *conn_request_sid* field of the newly allocated *struct open_request* object to the source socket SID of the message. If the *useclient* flag is set for the socket, then the *newconn_sid* field of the open request object is also set to this value. Otherwise, the *newconn_sid* field of the open request object is copied from the corresponding field of the socket. If *SYN* cookies are being used, then the open request object is discarded and recreated when the client's *ACK* is received. In this case, the *conn_request_sid* field is set to the SID of the *ACK* message.

When a TCP *ACK* is received for an existing *struct open_request* object, the *tcp_create_openreq_child* function sets the peer SID of the newly allocated *struct sock* object to the *conn_request_sid* field of the open request object, and it sets the SID of the new socket to the *newconn_sid* field of the open request object. The security class for the newly allocated *struct sock* object is copied from the listening socket. When a connection is accepted, the *inet_accept* function copies the socket SID and security class from the *struct sock* object into the inode for the *struct socket* object.

When a TCP *SYN-ACK* is received in the *SYN_SENT* state, the *tcp_rcv_state_process* function sets the peer SID of the client socket to the source socket SID of the message. When a *SYN* is received in the *SYN_SENT* state (a simultaneous open), the *tcp_rcv_state_process* function sets the peer SID of each socket to the source socket SID of the message.

For Unix stream sockets, the equivalent processing for connection establishment occurs entirely within the *unix_stream_connect* function. If the *useclient* flag is set on the listening socket, then the SID of the newly allocated *struct sock* object is set to the SID of the client socket. Otherwise, the SID of the new server socket is copied from the *newconn_sid* field of the listening socket. The peer SID of the client socket is set to the SID of the server socket, and the peer SID of the server socket is set to the SID of the client socket.

8.2.2 API extensions The Linux socket calls are implemented as library functions that invoke a single system call, *socketcall*, with a parameter that indicates the kind of call. Consequently, the extended socket calls were implemented simply by defining new call values to the *socketcall* system call. To permit the existing *fchsid* call to be used on sockets, the *inode_setattr* function was changed to call a new *sock_chsid* function if a socket is being relabeled.

Internally, the Linux network component uses an abstract interface to call the code specific to each protocol family. Extended forms of the *connect*, *accept*, *getname*, *listen*, *sendmsg* and *recvmsg* operations were added to the *struct proto_ops* structure to support the corresponding extended socket calls. A *chsid* operation was added to the *struct proto_ops* structure to support relabeling of sockets. An extended form of the *create* operation was added to the *net_proto_family* operations vector to support the *socket_secure* call. Within each protocol family, an abstract interface is used to call the transport layer protocol code. Extended forms of the *connect*, *sendmsg*, and *recvmsg* operations were added to the *struct proto* structure to support the corresponding extended socket calls.

The initialization function for ICMP (*icmp.c:__initfunc*) was modified to use the extended *create* operation to create the *icmp_socket* with the *icmp_socket* initial SID. Likewise, the initialization function for TCP (*tcp_ipv4.c:__initfunc*) was modified to create the *tcp_socket* with the *tcp_socket* initial SID.

The *inet_create* and *unix_create* functions were changed to set the SID of the socket when a particular SID is specified. The *inet_listen* and *unix_listen* functions were changed to set the *newconn_sid* field or the *useclient* field of the socket if the corresponding parameter was specified. The *udp_connect* and *unix_dgram_connect* functions were changed to set the peer SID of the socket. The *unix_dgram_sendmsg*, *ip_build_xmit*, and *ip_build_xmit_slow* functions were changed to set the destination socket SID and/or the message SID of the message if particular values were specified. The *tcp_v4_connect* function was changed to set the destination socket SID for the connection request message to the specified peer SID. The *tcp_v4_sendmsg* and *unix_stream_sendmsg* functions were changed to compare the specified message SID and/or destination socket SID with the actual values determined during connection establishment.

The *udp_deliver*, *raw_rcv_skb*, *unix_find_other* and *unix_dgram_sendmsg* functions were changed to compare the peer SID of the socket with the source socket SID of the message and to compare the SID of the socket with the destination socket SID of the message. The *tcp_v4_do_rcv* function was changed to compare the socket SID with the destination socket SID of the message when a message is received on a listening socket. If a mismatch occurs on a raw IP socket, then the packet is silently dropped. If a mismatch occurs on a UDP socket and the packet was to a unicast address, then an ICMP port unreachable message is sent in reply. If a mismatch occurs on a UDP socket and the packet was sent to a multicast or broadcast address, then the message is silently dropped. If a mismatch occurs on a listening TCP socket, then a TCP reset is sent in reply. If a mismatch occurs on a Unix domain socket, a connection refused error is returned to the connecting or sending process.

8.2.3 Control Requirements To minimize the overhead of permission checks, two AVC entry reference fields (*avcr* and *peer_avcr*) were added to the *struct sock* structure and one AVC entry reference field was added to the *struct device* structure. The *sk_alloc* function initializes these fields for new socket objects. The *devinet_ioctl* function initializes this field for devices when they are first accessed.

Since *acceptfrom* permission is initially checked by TCP when the open request object is created, an AVC entry reference field (*avcr*) was added to the *struct open_request* structure. This field is initialized when an open request object is created by the *cookie_v4_check* function or the *tcp_v4_conn_request* function. The field is set in these functions when it is used for the *acceptfrom* permission check.

To permit the *connectto* and *acceptfrom* permissions to be revalidated when traffic is sent or received on an established connection, a connection permission field (*conn_perm*) was also added to the *struct sock* structure. When a new TCP server socket is created, the *tcp_create_openreq_child* function sets *conn_perm* field to the *acceptfrom* permission, and it copies the *avcr* field from the open request object into the *peer_avcr* field. For client TCP sockets, the *tcp_rcv_state_process* function sets the *conn_perm* field to the *connectto* permission. The *peer_avcr* field is set in this function when it is used for the *connectto* permission check. For Unix stream sockets, the *conn_perm* and *peer_avcr* fields are set by *unix_stream_connect* for both the client socket and the server socket.

The control requirements implemented in each kernel function for TCP communication are shown in Table 32. Only the class and permission are shown for each control requirement; the source SID and target SID can be found in the corresponding design table. If *connectto* permission is denied during connection establishment, a connection refused error is returned to the local process and the socket is shut down. If *acceptfrom* or *newconn* permission is denied during connection establishment, a TCP reset is sent in reply to the connection request. The permission stored in the *conn_perm* field is revalidated by the *tcp_do_sendmsg* and *tcp_rcv_established* functions. If permission is no longer granted when *tcp_rcv_established* receives a message on a connection or when *tcp_do_sendmsg* attempts to send a message on a connection, then a connection reset error is returned to the local process and the socket is shut down. If *tcp_send*

FUNCTION(S)	CONTROL REQUIREMENT(S)	
	CLASS	PERM
inet_listen	socket socket	listen newconn
inet_stream_connect	socket	connect
inet_accept	socket	accept
inet_sendmsg	socket	write
inet_recvmsg	socket	read
tcp_v4_conn_request cookie_v4_check	socket socket	newconn acceptfrom
tcp_rcv_state_process	socket	connectto
tcp_do_sendmsg	socket socket	connectto acceptfrom
tcp_rcv_established	socket socket	connectto acceptfrom
ip_queue_xmit, ip_build_and_send_pkt, ip_forward, ipmr_queue_xmit	netif node	tcp_send tcp_send
ip_recv	netif node	tcp_recv tcp_recv

Table 32: Implementing the control requirements for TCP communication.

FUNCTION(S)	CONTROL REQUIREMENT(S)	
	CLASS	PERM
unix_listen	socket socket	listen newconn
unix_accept	socket	accept
unix_stream_recvmsg	socket	read
scm_detach_fds	fd	receive
unix_stream_connect	socket socket socket socket	connect connectto newconn acceptfrom
unix_stream_sendmsg	socket socket socket	write connectto acceptfrom

Table 33: Implementing the control requirements for Unix stream communication.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
inet_dgram_connect	socket	connect
inet_sendmsg	socket	write
inet_recvmsg	socket	read
udp_sendmsg, raw_sendmsg	socket socket	sendto send_msg
udp_deliver, raw_rcv_skb	socket socket	recvfrom recv_msg
ip_build_xmit, ip_build_xmit_slow, ip_forward	netif node	udp/rawip_send udp/rawip_send
ip_rcv	netif node	udp/rawip_recv udp/rawip_recv

Table 34: Implementing the control requirements for UDP or raw IP communication.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
unix_dgram_connect	socket	connect
unix_dgram_recvmsg	socket	read
scm_detach_fds	fd	receive
unix_dgram_sendmsg	socket socket socket socket socket	write sendto send_msg recvfrom recv_msg

Table 35: Implementing the control requirements for Unix datagram communication.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
inet_create, unix_create	socket	create
inet_bind, unix_bind	socket socket	bind name_bind
inet_getname, unix_getname	socket	getattr
sock_getsockopt inet_getsockopt	socket	getopt
sock_setsockopt inet_setsockopt	socket	setopt
inet_shutdown, unix_shutdown	socket	shutdown

Table 36: Implementing the control requirements for the other socket calls.

FUNCTION(S)	CONTROL REQUIREMENT(S) CLASS	PERM
unix_ioctl	socket	getattr
devinet_ioctl	netif netif	getattr setattr
ip_rt_ioctl	system	route_control
arp_ioctl	system	arp_control
rarp_ioctl	system	rarp_control
inet_ioctl	system	net_io_control

Table 37: Implementing the control requirements for ioctl commands.

or *tcp_recv* permission is denied, then an ICMP port unreachable message is sent if the message was locally generated or an ICMP host unreachable message is sent if the message is being forwarded.

Table 33 shows the control requirements implemented in each kernel function for Unix stream communication. If *newconn*, *acceptfrom*, or *connectto* permission is denied during connection establishment, then a connection refused error is returned to the connecting process. If *acceptfrom* or *connectto* permission is no longer granted when data is sent on the connection, then a connection reset error is returned to the sending process and the socket is shut down. If *receive* permission is not granted for an open file description, then the descriptors and any subsequent descriptors in the message are discarded.

Table 34 shows the control requirements implemented in each kernel function for UDP or raw IP communication. If *recvfrom* or *recv_msg* permission is denied when a UDP unicast message is received, then an ICMP port unreachable messsage is sent in reply. If either of these permissions are denied for a UDP multicast or broadcast message or a raw IP message, then the message is silently dropped. If *udp_send* or *rawip_send* permission is denied, then a permission denied error is returned to the local process if the message was locally generated or an ICMP host unreachable message is sent if the message is being forwarded. If *udp_recv* or *rawip_recv* permission is denied for a unicast message, then an ICMP port unreachable message is sent.

Table 35 shows the control requirements implemented in each kernel function for Unix datagram communication. If *recvfrom* or *recv_msg* permission is denied when a message is sent, then a connection refused error is re-

OBJECT CLASS
semaphore set
message queue
message
shared memory

Table 38: Object classes for the Linux System V IPC component.

PERMISSION(S)	DESCRIPTION
associate	Associate a key with a semaphore set
read	Read
write	Write
create	Create
destroy	Destroy
getattr	Get attributes
setattr	Set attributes

Table 39: Permissions for the semaphore object class.

PERMISSION(S)	DESCRIPTION
associate	Associate a key with a queue
read	Read
write	Write
create	Create
destroy	Destroy
getattr	Get attributes
setattr	Set attributes
enqueue	Message may reside on queue

Table 40: Permissions for the message queue object class.

turned to the sending process. If *receive* permission is not granted for an open file description, then the descriptors and any subsequent descriptors in the message are discarded.

The implementation of the control requirements for the other socket calls is shown in Table 36. The *inet_bind* function only checks *name_bind* permission if the port number is outside of the range used to automatically bind sockets. The *unix_bind* function only checks *name_bind* permission if the name is in the file system namespace. Table 37 shows the implementation of the control requirements for the ioctl commands.

9 System V IPC

This section describes the design and implementation of the Flask security mechanisms for Linux System V Inter-Process Communication.

9.1 Design

This section describes our design for integrating the Flask security mechanisms into the Linux System V IPC component. It begins with a discussion of the object classes and permissions defined for the component. This is followed by a description of the control requirements for the system calls used to manage and perform IPC operations. Finally, the new IPC-related system calls defined for security-aware applications are described.

9.1.1 Object Classes The set of object classes for the System V IPC component is shown in Table 38. A class is defined for each System V IPC abstraction. Additionally, a class is defined for individual messages within a message queue, so that messages can be individually labeled and controlled. By default, the SID of a System V IPC object will be set to the SID of the creating process. For System V message queues, an additional SID attribute will be bound to each message. That SID is set by default to the SID of the message sending process.

9.1.2 Permissions The permissions defined for controlling access to each System V IPC object class are shown in Table 39, Table 40, Table 41, and Table 42.

The *semget*, *msgget*, and *shmget* system calls and their associated *_secure* variants each require *associate* access to associate a key with an ID when accessing an existing object by key. If an object is being created and the process has *create* permission, it is assumed that it implicitly has the ability to associate a key with the created IPC object and *associate* permission is not required. Since the key can also be obtained via *semctl*, *msgctl*, and *shmctl* calls with the SEM_STAT, MSG_STAT, SHM_STAT, and IPC_STAT options these also require *associate* permission.

The *semctl*(IPC_INFO), *semctl*(SEM_INFO), *msgctl*(IPC_INFO), *msgctl*(MSG_INFO), *shmctl*(IPC_INFO) and *shmctl*(SHM_INFO) system calls return information to the calling process that is combined from all objects of the given type without regard to the object SID. The design therefore restricts use of these calls to processes with a new system permission

PERMISSION(S)	DESCRIPTION
send	Add a message to a queue
receive	Remove a message from a queue

Table 41: Permissions for the message object class.

PERMISSION(S)	DESCRIPTION
associate	Associate a key with a segment
read	Read
write	Write
create	Create
destroy	Destroy
getattr	Get attributes
setattr	Set attributes
lock	(Un)lock page(s) in memory

Table 42: Permissions for the shared memory object class.

(*ipc_info*.) This access is now necessary for correct functioning of the *ipcs* program. It may be desirable in the future to modify the *ipcs* program to selectively return information based on the executing SID.

9.1.3 Control Requirements The control requirements for the System V IPC system calls are shown in Table 43, Table 44, and Table 45. For the most part these are obvious extensions of the standard Linux permission bits.

Unlike for the file system *open* call, *semget*, *msgget*, *shmget* and their *_secure* variants can not validate the requested *read* or *write* permissions since these system calls may also be used to obtain an id for use in the *semctl*, *msgctl*, and *shmctl* calls which may require neither *read* nor *write* permission to be successful. The *read* and *write* permissions are checked when making system calls which perform the actual data operations.

The *message queue* object employs the *read* and *write* permission to determine if a specific process may utilize a specific queue for receiving or sending messages, respectively. In contrast the *message* object employs *send* and *receive* permissions to determine if a given process is allowed to receive or send a given message. The *enqueue* permission is used to determine if a given message is permitted to "pass through" the given queue.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
semget, semget_secure	sem sem	create associate	current current	sem sem
semsid	sem	getattr	current	sem
semop (sem_op==0)	sem	read	current	sem
semop (sem_op!=0)	sem	write	current	sem
semctl.SEM_STAT, IPC_STAT	sem sem	getattr associate	current current	sem sem
semctl.IPC_GETNCNT, IPC_GETPID, IPC_GETZCNT	sem	getattr	current	sem
semctl.IPC_SET	sem	setattr	current	sem
semctl.IPC_RMID	sem	destroy	current	sem
semctl.IPC_GETALL, IPC_GETVAL	sem	read	current	sem
semctl.IPC_SETALL, IPC_SETVAL	sem	write	current	sem
semctl.SEM_INFO, IPC_INFO	system	ipc_info	current	kernel

Table 43: Control requirements for manipulating semaphores.

	CONTROL REQUIREMENT(S)			
CALL(S)	CLASS	PERM	SSID	TSID
msgget, msgget_secure	msgq msgq	create associate	current current	msgq msgq
msgsid	msgq	getattr	current	msgq
msgsnd, msgsnd_secure	msgq msg msgq	write send enqueue	current current msg	msgq msg msgq
msgrcv, msgrcv_secure	msgq msg	read receive	current current	msgq msg
msgctl.MSG_STAT, IPC_STAT	msgq msgq	getattr associate	current current	msgq msgq
msgctl.IPC_SET	msgq	setattr	current	msgq
msgctl.IPC_RMID	msgq	destroy	current	msgq
msgctl.MSG_INFO, IPC_INFO	system	ipc_info	current	kernel

Table 44: Control requirements for manipulating message queues.

CALL(S)	CONTROL REQUIREMENT(S) CLASS	PERM	SSID	TSID
shmget, shmget_secure	shm shm	create associate	current current	shm shm
shmsid	shm	getattr	current	shm
shmat (SHM_RDONLY)	shm	read	current	shm
shmat (!SHM_RDONLY)	shm shm	read write	current current	shm shm
shmctl.SHM_STAT, IPC_STAT	shm shm	getattr associate	current current	shm shm
shmctl.IPC_SET	shm	setattr	current	shm
shmctl.IPC_RMID	shm	destroy	current	shm
shmctl.SHM_LOCK, SHM_UNLOCK	shm	lock	current	shm
shmctl.SHM_INFO, IPC_INFO	system	ipc_info	current	kernel

Table 45: Control requirements for manipulating shared memory.

9.1.4 API extensions

New calls were added to permit a process to create a System V IPC object with a specified SID and to permit a process to obtain the SID of an existing object. New calls were also added to permit a sending process to specify a particular SID for a message and to permit a receiving process to obtain the SID of the received message.

The *semsid*, *msgsid*, and *shmsid* system calls were added to retrieve the security ID for a given *semaphore set*, *message queue*, or *shared memory* segment respectively. The *semget_secure*, *msgget_secure*, and *shmget_secure* system calls were added to provide a way to request an IPC object with a specific security ID.

The *msgsnd_secure* and *msgrcv_secure* system calls were added to enable sending and receiving a message with a specific SID to or from a message queue. The *msgrcv_secure* call takes a SID reference parameter which is either a specific SID to use in selecting a message, or a null or wildcard SID to allow the return of a message at any SID to which access is permitted. The call also returns the SID of the returned message using this reference.

9.2 Implementation

This section describes our implementation for integrating the Flask security mechanisms into the Linux System V IPC component.

9.2.1 Labeling

New structures (*semid_kernel* and *msqid_kernel*) were added to parallel (and contain) *semid_ds* and *msqid_ds* respectively. This permits the kernel to maintain SIDs in the data structures without altering the public versions of each structure and the thereby the existing API. A number of small changes were made in *sem.h*, *msg.h*, *sem.c* and *msg.c* to accommodate the new structures. These are similar to the existing *shmid_kernel* structure which is used to maintain kernel mappings for memory segments. The addition of label data to each of these three structures is controlled by conditional compile directives, however, the new kernel structures themselves were added to the base code as were the related structure references.

9.2.2 API Extensions

In reality no new system calls were added, although eight new APIs were added to the system call interface. The new APIs utilize the previously existing *sys_ipc* system call demultiplexer in *sys_i386.c* to dispatch the new IPC related calls. Currently only the Intel 32-bit architecture has been modified.

The *semget_secure*, *msgget_secure*, and shmget_secure calls allow a caller to request an IPC object with a given key and a given SID. If the caller has *create* access but the key is already allocated to an IPC object with a different SID, the caller is returned EACCES.

The *msgrcv* and *msgrcv_secure* calls skip messages that are labeled with a SID for which the calling process does not have *receive* access rather than return EACCES. This may result in access checks being performed and denial of access being audited for messages whose existence is not even apparent to the calling process when a) the *msgrcv* call is used, or the *msgrcv_secure* call is used with a SID of SECSID_WILD or SECSID_NULL, and b) messages exist on the queue for which the process does not have *receive* permission, and c) access has been revoked from the process for the queue or from messages to which it had access when it called *msgrcv* or *msgrcv_secure*. In that (unusual) case, the calling process will also wait indefinitely for a message.

A beneficial implication of the interactions between the *read*, *write*, *receive*, *send*, and *enqueue* permissions and independent SIDs for *message queues* and individual *messages* is that policies may be developed which allow restricted multi-level message queues. These may be desired to create a convenient one-way communication mechanism for processes. Policy writers, of course, need to carefully consider the the implications of such a one-way channel if it is desired and implemented.

9.2.3 Control Requirements The *semctl*, *msgctl* and *shmctl* system calls with the IPC_RMID or IPC_SET options each allowed a process with CAP_SYS_ADMIN to bypass the standard Linux owner checks. The Flask checks, however, are not bypassed by CAP_SYS_ADMIN, and as described in Section 5.2, the use of CAP_SYS_ADMIN even to bypass the standard owner checks is subject to additional Flask checks.

Each control point that has the potential to generate an audit record adds the ID of the semaphore set, message queue, or shared memory region to the audit data included in the record.

The *shmat*(!SHM_RDONLY) control point is implemented as a check for *read* and *write* since the logical semantics as well as the x86 implementation of not-readonly are read and write.

Since *shmat* also writes the access granted into the memory segment structures for hardware use on each access, a call-back will eventually be necessary to allow the access control embodied in the memory structures to be revalidated or revoked on a policy change.

The case statement conditions in the *sys_semctl* function were slightly reordered to collect some *read* based access checks and *getattr* access checks that were both covered by the standard Linux read permission bit. A seemingly redundant check was found for the "ID Removed" (EIDRM) condition between switch statements in that function that was already covered by the check at the beginning of *sys_semctl*. The first case in the switch following the EIDRM check is a permission check for GETALL that also seems to be redundant with the check previously performed in the cases above. No control point was added for the second permission check as the previous permission check should still be valid. (There are no process sleeps in the GETALL code path between these points.) No other changes were made to improve the flow of the case statements, as this was seen as counter-productive given the reorganization in later kernel versions.

The *sem_exit* function performs a check when freeing undo structures that the semaphore values do not become negative. A printk was added after that check to allow detect failure of that assumption. Since the printk should never be executed unless there is a kernel bug, there should be no impact from this change.

10 System Call Review

This section contains the results of a review of the Linux/i386 system call interface to determine what additional Flask controls are needed. The following subsections describe the results for each category of system calls. As controls recommended in this review are implemented, the corresponding discussion should be removed from this section and integrated into an appropriate section of the document.

10.1 Process Management

This subsection describes the results of the system call review for calls related to process management.

10.1.1 Scheduling Flask provides controls over the ability to observe or modify the scheduling characteristics of other processes and the ability to increase priority. However, Flask does not yet provide a scheduling policy. Support for a new scheduling policy based on the security contexts of processes would be desirable. Such support will likely require a new interface to the security server.

10.1.2 Sessions and Process Groups The ability to observe or set the session and process group identifiers of other processes is controlled by Flask, but the session and process group abstractions are not labeled or controlled. The Flask mechanisms could be used to provide a label for each session and process group, to control what processes may be in each process group, and to control what terminal may serve as the controlling terminal for each process group. However, it is not clear that such labeling and controls are needed, since signal delivery and terminal access are controlled through the existing Flask signal and file controls. Further study of the use of session and process groups is needed.

10.1.3 User and Group Identity The *setuid* , *seteuid* , *setreuid* , *setresuid* , and *setfsuid* calls may be used to set the user identity attributes of the calling process. The *setgid* , *setegid* , *setregid* , *setresgid* , *setfsgid* , and *setgroups* calls may be used to set the group identity attributes of the calling process. Linux permits unprivileged processes to perform certain kinds of changes to their identity attributes, such as changing the effective identity to the real identity or vice versa, or changing the effective identity to the saved identity. Linux permits more general changes in identity for processes that have the *CAP_SETUID* and *CAP_SETGID* capabilities.

These calls only affect the private state of the calling process. Furthermore, the Flask controls are not based on the Linux identity attributes. Consequently, changes in Linux identity attributes are irrelevant to the Flask security policy and do not need to be controlled by the policy. However, it may be valuable to provide Flask controls on these calls to allow the policy to confine changes in Linux identity. The Flask *cap_setuid* and *cap_setgid* permissions are checked when the corresponding capabilities are required by Linux. If it is desirable for the policy to be able to confine Linux identity changes, then new Flask permissions need to be defined to control all uses of these system calls.

10.1.4 Capabilities The ability to get or set the capability sets of another process is controlled by Flask. Flask controls the use of capabilities by requiring that a process also have a corresponding capability permission. Hence, possession of a capability is necessary but not sufficient to have the corresponding privilege when Flask is enabled. Since Flask directly controls the use of capabilities, it is not necessary to control the setting and inheritance of capability sets other than ensuring that the ability of a process to observe or change the capability sets of another process is controlled.

Flask could be extended to provide a finer-grained replacement mechanism for capabilities. Such a mechanism was developed for one of Flask's predecessors, the DTOS system. This mechanism permitted privileges to be granted based on both the attributes of the process and the attributes of the relevant object, *e.g.* discretionary read override could be granted to a particular set of files. Since the mechanism obtained privilege decisions from the Flask security server, management of privileges was centralized and verification that privileges were granted appropriately was straightforward.

10.1.5 Timers The *alarm* and *setitimer* calls may be used to arrange for a signal to be sent to the calling process after an interval. The *getitimer* call obtains the value of an interval timer. The calls are implemented in *kernel/sched.c* and *kernel/itimer.c*. Currently, no controls are performed. A process could arrange for a signal to be delivered and perform an *execve* or *execve_secure* before the signal is generated, thus effectively delivering a signal to itself after a SID change. This signal is not subject to any access checking, so additional controls are necessary when the *execve* is performed. Timers could be cleared upon an *execve* that changes SID if the calling process lacks the appropriate signal permission to the transformed process.

10.1.6 Resource Limits and Usage The *setrlimit* call may be used to change the resource limits for the calling process. The call is implemented in *kernel/sys.c*. Linux requires the *CAP_SYS_RESOURCE* capability to use the *setrlimit* call to increase the soft or hard limit above the current hard limits, so Flask requires *cap_sys_resource* under the same conditions. Flask does not provide a process resource limit policy. Support for defining resource limits based on the security contexts of processes would be desirable. For now, Flask only controls the ability to increase the limit above the current hard limit.

The *getrusage* call may be used to get resource usage information for the calling process and its children. Similarly, the *times* call may be used to obtain the time usage of the calling process and its children. Linux does not control the use of these calls. Since the usage statistics of a child process are only added into the parent's combined statistics for its children if the parent reaps the child, the existing Flask wait controls are sufficient. No other controls seem necessary.

10.1.7 Other Process Calls The *prctl* call is a general interface for performing operations on a process. It only supports a single operation that sets or clears the signal that the calling process will receive when its parent dies. The existing Flask controls for signals ensure that the delivery of the signal is controlled. However, it would be useful to check the appropriate signal permission when this call is used so that the calling process will be aware of any permission failure.

The *rt_sigqueueinfo* call is a variant of *kill* for real-time signals. This call is already controlled through the existing Flask signal permissions.

The *getpid* and *getppid* calls may be used to obtain the process identifier of the calling process and its parent, respectively. The *getpid* call does not require any controls, since it only reveals private state of the calling process. The *getppid* call would only need to be controlled if it would be useful to conceal the PID of the parent, but such a need is not evident.

10.2 Memory Management

This subsection describes the results of the system call review for calls related to memory management.

The *mprotect* call may be used to set the protection on a region of memory. The call is implemented in *mm/mprotect.c*. Linux requires that the new protection be a subset of the maximum protection on the mapping. For anonymous memory or a private copy-on-write mapping of a file, the maximum protection allows all accesses. For a shared mapping, the maximum protection always allows read and execute access but only allows write access if the file is open for writing.

Flask ensures that *mprotect* cannot be used to increase the current protection on memory-mapped files beyond what the security policy authorizes. Flask should also control the ability to execute anonymous memory. A new permission could be introduced based on the SID of a process that controls whether the process is allowed to execute anonymous memory.

The *mlock* and *munlock* calls may be used to disable and reenable paging for a range of memory. The *mlockall* and *munlockall* calls may be used to disable and reenable paging for all pages mapped into the address space of the calling process. These calls are implemented in *mm/mlock.c*. Linux requires *CAP_IPC_LOCK* to disable paging, so Flask requires *cap_ipc_lock* permission. No additional controls seem necessary.

10.3 File System

This subsection describes the results of the system call review for calls related to the file system.

The *nfsservctl* call is the interface to the kernel NFS daemon. The call is implemented in *fs/nfsd/nfsctl.c*. Linux requires *CAP_SYS_ADMIN* to use the call, so Flask requires *cap_sys_admin* permission. Since the Flask controls have not yet been integrated into the Linux NFS implementation, no further controls are required at this time. Separate permissions for the individual operations may be introduced at a later time.

The *quotactl* call may be used to manipulate disk quotas. This call is implemented in *fs/dquot.c*. Linux requires *CAP_SYS_RESOURCE* for enabling or disabling quotas, getting the quota of another user or group, or setting a quota. Hence, Flask requires *cap_sys_resource* permission for these commands. When enabling quotas, a quota file is specified. This file must already exist, typically being created by the *quotacheck* program. The file is opened for read and write access, and the existing Flask file access controls are applied. It might be useful to add a new permission controlling what files may be used as quota files. Linux does not control the *quotactl* commands for syncing the quota files, obtaining quota-related statistics, or obtaining the quota limits and current usage for user or group of the calling process. Flask does not provide a disk quota policy. Support for defining disk quotas based on the security contexts of processes would be desirable.

The *bdflush* call may be used to start, flush or tune the buffer-dirty flush daemon. The call is implemented in *fs/buffer.c*. Linux requires the *CAP_SYS_ADMIN* capability, so Flask requires *cap_sys_admin* permission. It might be useful to add new permissions to control the individual operations provided by the call.

The *swapon* and *swapoff* calls may be used to start and stop swapping to a file or device. Linux requires *CAP_SYS_ADMIN* and search access to the file to use either call. Flask requires *cap_sys_admin* permission and *search* permission. It might be useful to add a new permission controlling what files may be used as swap files.

The *chroot* system call may be used to change the root directory. The call is implemented in *fs/open.c*. Linux requires search access to the new root directory and *CAP_SYS_CHROOT*. Flask requires *search* permission to the new root directory and *cap_sys_chroot* permission. No further controls seem to be necessary.

10.4 Kernel Modules

This subsection describes the results of the system call review for calls related to kernel modules.

The *create_module* call may be used to register a name and to reserve kernel memory for a loadable module. The *init_module* call may be used to load a relocated module image into kernel memory and to run the module's initialization function. The *delete_module* call may be used to remove modules. These calls are implemented in *kernel/module.c*. Linux requires *CAP_SYS_MODULE* to use any of these three calls, and Flask requires the corresponding *cap_sys_module* permission. No additional controls seem to be necessary for these calls.

The *query_module* call may be used to obtain information related to loadable modules. The *get_kernel_syms* call may be used to obtain the kernel and modules symbols. This call is obsoleted by *query_module*. Linux does not control the use of these two calls. Flask controls should be defined for these calls to control their use.

The kernel module loader (*kernel/kmod.c*) runs modprobe to automatically load modules when they are requested. The kernel module loader runs as the superuser with all capabilities enabled. The kernel module loader was changed for Flask to run with the *kmod* initial SID. Otherwise, the Flask controls would be based on the SID of the user process.

Although the Flask controls for module-related calls are straightforward, protection of the kernel module facility requires configuration of the security policy to label and control access to the module object files, the module utilities, the module configuration files, and the kernel path for modprobe. The policy configuration for kernel modules is described in [7].

10.5 System Operations

This subsection describes the results of the system call review for calls related to the overall system.

The *stime* and *settimeofday* system calls may be used to set the system time and date. Both calls are implemented in *kernel/time.c*. Linux requires *CAP_SYS_TIME* to use these calls, so a process must have the Flask *cap_sys_time* permission to use these calls. No further controls seem to be necessary. The *adjtimex* call may be used to read or modify the clock adjustment parameters. This call is implemented in *kernel/time.c*. Linux requires *CAP_SYS_TIME* to use this call to modify the parameters, so Flask requires *cap_sys_time*. No further controls seem necessary.

The *sethostname* and *setdomainname* calls may be used to set the host and domain names for the system. Linux requires *CAP_SYS_ADMIN* to use either call. Flask requires *cap_sys_admin* permission. No other controls seem necessary.

The *acct* call may be used to enable or disable process accounting. The call is implemented in *kernel/acct.c*. Linux requires the calling process to have *CAP_SYS_PACCT* to use the call. If the call is used to set the accounting file, then the calling process must also be able to open the accounting file with append access. The Flask *cap_sys_pacct* permission is checked when the call is used, and the Flask file mandatory access controls are checked if an accounting file is specified. It might be useful to add a new permission controlling what files may be used for accounting.

The *reboot* call may be used to reboot the system or to enable or disable the reboot keystroke. The call is implemented in *kernel/sys.c*. Linux requires that the calling process have *CAP_SYS_BOOT*, so Flask requires that the calling process have *cap_sys_boot* permission. No other controls seem necessary.

The *ioperm* call may be used to set I/O port access permission bits for the calling process for a specified port and range. The call may only be used for the first 0x3ff I/O ports. This call is implemented in *arch/i386/kernel/ioport.c*. If the permission bits are being set to anything non-zero, then Linux requires that the calling process have *CAP_SYS_RAWIO*. Thus, Flask requires the corresponding *cap_sys_rawio* permission. Port access permissions are not inherited on *fork* but they are inherited across *execve*. New Flask controls should be defined to control inheritance of port access permissions. It may also be desirable to support individual labeling of different I/O ports and to add a permission controlling access to particular ports.

The *iopl* call may be used to change the I/O privilege level of the calling process. The call is necessary for more ports than 0x3ff. For example, the call is used by 8514-compatible X servers to run under Linux. This call is implemented in *arch/i386/kernel/ioport.c*. If the privilege level is being increased, then Linux requires that the calling process have *CAP_SYS_RAWIO*. Thus, Flask requires the corresponding *cap_sys_rawio* permission. The privilege level is inherited on *fork* and across *execve*. New Flask controls should be defined to control inheritance of I/O privilege levels. It may also be desirable to add separate permissions for the different levels.

The *syslog* call may be used to read or clear the kernel message ring buffer and to set the console log level. Linux requires that the calling process have *CAP_SYS_ADMIN* to use any syslog operation except for the operation to read the last 4k of messages in the ring buffer. Hence, Flask requires *cap_sys_admin* permission for all of the other operations. Separate Flask permissions should be defined for the different kinds of syslog operations, *e.g.* separate permissions to control reading and clearing the last 4k of messages versus changing the console log level. Additionally, it seems desirable to control the ability of processes to read the last 4k of messages, so a Flask permission should be added to control this operation.

The *sysinfo* call may be used to obtain information on overall system statistics such as the load average, available memory, and number of current processes. Linux does not control the ability to use this call. A Flask permission should be defined to control the use of this call.

11 To Do

This section lists tasks that have not yet been completed for integrating the Flask security mechanisms into the Linux kernel.

- Add controls identified by system call review
- Perform functional and performance testing
- Add mandatory controls for System V IPC
- Integrate IPSEC with network mandatory controls
- Add mandatory controls for NFS
- Add support for polyinstantiated directories
- Add support for polyinstantiated ports
- Add notifications for completed operations
- Add policy change callbacks
- Integrate file cryptography with file mandatory controls
- Replace SIDs with SID descriptors (reference-counted SIDs)

References

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

[2] T. Fine and S. E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[3] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1996. Std 1003.1, 1996 Edition.

[4] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Oct. 1998.

[5] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[6] O. S. Saydjari, J. M. Beckman, and J. R. Leaman. LOCK Trek: Navigating Uncharted Space. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 167–175, 1989.

[7] S. Smalley and T. Fraser. A Security Policy Configuration for the Security-Enhanced Linux. Technical report, NAI Labs, Oct. 2000.

[8] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, Aug. 1999.