CHAPTER 9

Case Study: Building a Secure Operating System for Linux

The Linux operating system is a complete reimplementation of the POSIX interface initiated by Linus Torvalds [187]. Linux gained popularity throughout the 1990s, resulting in the promotion of Linux as a viable alternative to Windows, particularly for server systems (e.g., web servers). As Linux achieved acceptance, variety of efforts began to address the security problems of traditional UNIX systems (see Chapter 4). In this chapter, we describe the resulting approach for enforcing mandatory access control, the Linux Security Modules (LSM) framework. The LSM framework defines a reference monitor interface for Linux behind which a variety of reference monitor implementations are possible. We also examine one of the LSM reference monitors, Security-enhanced Linux (SELinux), and evaluate how it uses the LSM framework to implement the reference monitor guarantees of Chapter 2.

9.1 LINUX SECURITY MODULES

The Linux Security Modules (LSM) framework is a reference monitor system for the Linux kernel [342]. The LSM framework consists of two parts, a reference monitor interface integrated into the mainline Linux kernel (since version 2.6) and a reference monitor module, called an LSM, that implements reference monitor function (i.e., authorization module and policy store, see Chapter 2 for the reference monitor concept definition) behind that interface. Several independent modules have been developed for the LSM framework [228, 183, 127, 229] to implement different forms of reference monitor function (e.g., different policy modules). We examine the LSM reference monitor interface in this section, and one of its LSMs, Security-Enhanced Linux (SELinux) [229] in the subsequent section. There are other LSMs and a debate remains over which LSM approach is most effective, but SELinux is certainly the most comprehensive of the LSMs.

9.1.1 LSM HISTORY

In the late 1990s, the Linux operating system gained the necessary support to make it a viable alternative in the UNIX system market. Although there were a variety of UNIX variants, such as AIX and HP/UNIX, and even other open source systems, such as the BSD variants, Linux became the mindshare leader among UNIX systems. Large server vendors, such as IBM and HP, threw their support behind Linux, and it soon became the main competitor to Microsoft Windows.

Also, in the late 1990s, a number of projects emerged that retrofit various security features into the Linux kernel. Since Linux was open source, anyone could modify it to meet their requirements (as long as they released their code back to the community, per the GNU Public License requirements [112]). A variety of prototype systems emerged, including Argus PitBull [13], LIDS [183], AppArmor (originally called Subdomain) [228], RSBAC [240], GRSecurity [296], DTE (see Chapter 7) for Linux [127], Medusa DS9 [204], OpenWall [236], HP's Secure OS Software for Linux [80], and a retrofit of the former Flask/DTOS/DTMach system (see Chapter 7), now called SELinux. All these modified Linux systems varied in fundamental ways, but all aimed to provide a valuable security function. AppArmor and PitBull were both sold as commercial products.

In 2001, momentum was growing for inclusion of a reference monitor in the Linux kernel. Problems with worms, viruses, and denial-of-service attacks were reaching a significant level, although mostly on the Windows platform. At the Linux kernel summit that year, the SELinux prototype was presented, and the Linux community, including Linus Torvalds in particular, seemed to accept the idea that a reference monitor was necessary. However, Linus faced two challenges. First, he was not a security expert, so he could not easily decide among the approaches and felt it was not appropriate for him to make such a decision. Second, the security community itself could not agree on a single, "best" approach, so Linus could not depend on the security community to guide him to a single approach. As a result, Linus argued for a design based on kernel modules where a single interface could support all the necessary modules. This approach became the LSM framework.

A community formed around the idea of building a single reference monitor interface for Linux (although not all the Linux security prototype researchers agreed [239, 297] [1]), and this community designed and implemented the LSM framework. The main task was to implement the LSM reference monitor interface. The design of the LSM framework's reference monitor interface had the following goals [342]:

- The reference monitor interface must be truly generic, such that "using a different security model is merely matter of loading a different kernel module"
- The reference monitor interfaces must be "conceptually simple, minimally invasive, and efficient"
- Must support the POSIX.1e capabilities mechanism as an "optional security module"

The first two requirements motivated collecting the union of the authorization queries from all previous Linux security, such that all modules could be supported, but restricting the number of authorization queries as much as possible to prevent redundant authorizations that made add complexity and impact performance. While the LSM interface was designed manually [342], source code analysis tools were built to verify the completeness [351] and consistency [149] of the LSM interface, finding six interface bugs that were resolved.

Performance analysis showed that the most of the LSM interface had no tangible performance impact [342], but the CIPSO implementation (i.e., labeled networking, see Section 7.5.2) provided

[1]The RSBAC comment dated April 2006 that LSM would be removed from the official kernel is no longer current. Linus Torvalds reaffirmed his support for LSM in the 2006 Linux Kernel Summit, and LSM will remain part of the mainline Linux kernel for the foreseeable future.

with the initial LSM interface was rejected. The performance overhead of keeping labels consistent under packet fragmentation and defragmentation, even if no security policy was being enforced, was too costly. Two other alternatives for labeled networking are now supported by the Linux kernel. First, Labeled IPsec [148], based on the Flask labeled networking [50], negotiates labels for IPsec network communications in addition to cryptographic parameters. An LSM controls network communication by authorizing whether a process can use a particular IPsec communication channel. Since the label of the IPsec channel is established at negotiation time, there is no need to include the label in the packet. Second, Paul Moore built a new implementation of CIPSO for Linux, called Netlabel [214]. Netlabel provides a less intrusive version of CIPSO which was accepted by the Linux community.

The Linux Security Modules framework was officially added to the Linux kernel with the release of version 2.6. The SELinux module and a module for implementing POSIX capabilities [307] were included with the release of LSM. Novell, the distributor of SuSE Linux, purchased the company that supported AppArmor, so SuSE and other Linux distributions support the AppArmor LSM as well.

SELinux and AppArmor have become the major LSMs. While both provide tangible Linux security improvements, converting Linux (or any UNIX system) to a system that can satisfy reference monitor guarantees is a difficult task. However, with Linus reaffirming his support for the LSM framework [188] in 2006 and a variety of Linux vendors support security behind LSMs, the LSM framework can be considered a modest success.

9.1.2 LSM IMPLEMENTATION

The LSM framework implementation consists of three parts: (1) the reference monitor interface definition; (2) the reference monitor interface placement; and (3) the reference monitor implementations themselves.

LSM Reference Monitor Interface Definition The LSM interface definition specifies the ways that the Linux kernel can invoke the LSM reference monitor. Linux header file `include/linux/security.h` lists a set of function pointers that invoke functions in the loaded LSM. A single structure, called `security_operations`, contains all these LSM function pointers. We refer to these function pointers collectively as the *LSM hooks*. Fundamentally, the LSM hooks correspond to LSM authorization queries, but the LSM interface must also include LSM hooks for other LSM tasks, such as labeling, transition, and maintenance of labels.

Two examples of LSM hooks are shown below.

```
static inline int security_inode_create (struct inode *dir,
                                         struct dentry *dentry,
                                         int mode)
{
        if (unlikely (IS_PRIVATE (dir)))
                return 0;
        return security_ops->inode_create (dir, dentry, mode);
}

static inline int security_file_fcntl (struct file *file, unsigned int cmd,
                                       unsigned long arg)
{
        return security_ops->file_fcntl (file, cmd, arg);
}
```

First, `security_inode_create` authorizes whether a process is permitted by the LSM to create a new file, indicated by `dentry`, in a particular directory `dir`. The LSM hook is invoked through the call to the function pointer defined by `security_ops->inode_create`. The LSM loaded defines how the authorization is performed. Second, `security_file_fcntl` authorizes a specified process's ability to invoke `fcntl` on a specific file. Subsequent LSM hooks in the function `do_fcntl` enable an LSM to limit certain uses of `fcntl` independently (e.g., setting the `fowner` field that signals the associated process on file operations).

In all there are over 150 LSM hooks that enable authorizations (as above), and the other LSM operations of labeling, transitioning labels, and maintenance of labels. While different LSM hooks are intended to serve different purposes, they all have a similar format to the two listed above.

LSM Reference Monitor Interface Placement The main challenge in the design of the LSM framework is the placement of the LSM hook. Most of the LSM hooks are associated with a specific system call, so for these the LSM hook is placed at the entry to the system call. However, several of the LSM hooks cannot be placed at the system call entry point (e.g., to prevent TOCTTOU attacks, see Chapter 2). For example, as shown in Figure 9.1, the `open` system call converts a file path to a file descriptor that enables access (i.e., read and/or write) to the associated file. Locating the specific file described by the file path requires authorizing access to the directories that accessed along the path, any file links in the path, and finally authorizing to the target file for the specific operations requested. Since these components are extracted from the file path at various points in the `open` processing, the LSM hook placement is nontrivial.

While there are some discretionary checks in place that guided the placement of the LSM hooks for `open`, the process by which the LSM hooks were placed was largely ad hoc. For ones where no previous discretionary authorization was performed, the implementors made a manual placement. Some placements were found to be wrong, and some security-sensitive operations were found to be lacking mediation, but these issues were resolved through source code analysis [351, 149].

An LSM hook is placed in the code using the inline function declarations (e.g., `security_inode_create`) which is expanded at compile-time to the LSM hooks as shown by the code above. Inline functions for each LSM hook are used to improve the readability of the code.

LSM Reference Monitor Implementations Finally, LSMs must be built to perform the actual authorizations. Actual LSMs include AppArmor [228], the Linux Intrusion Detection System

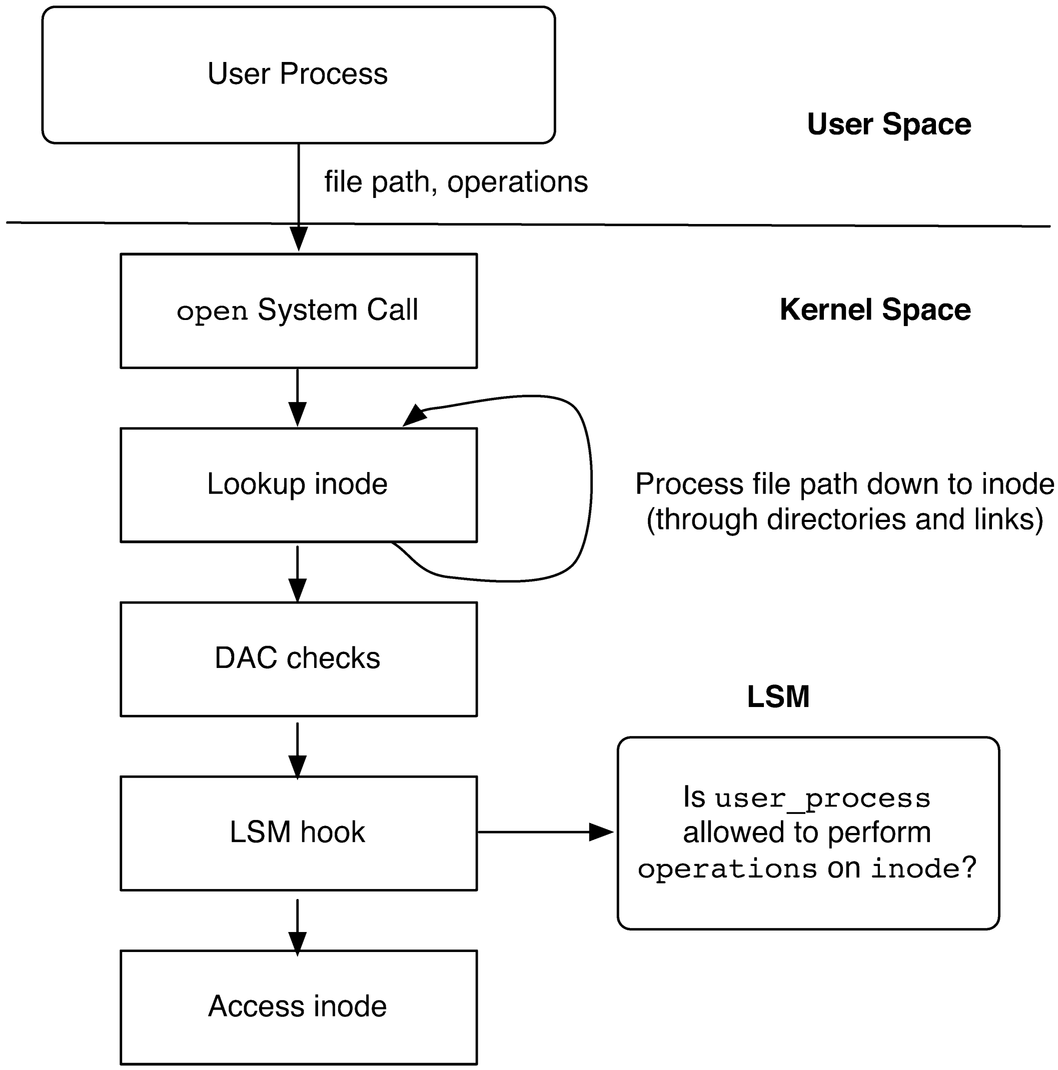
User Process
User Space
file path, operations
open System Call
Kernel Space
Lookup inode
Process file path down to inode (through directories and links)
DAC checks
LSM
LSM hook
Is user_process allowed to perform operations on inode?
Access inode

Figure 9.1: LSM Hook Architecture

(LIDS) [183], SELinux [229], and POSIX capabilities [307]. Each of LSMs provide a different approach to mandatory access control, excepting POSIX capabilities which was an existing discretionary mechanism in Linux. POSIX capabilities were converted to a module to enable independent development from the mainline kernel and because some LSMs aimed to implement the capability controls in a different manner [342].

Although Gasser and Schell identified that different security kernel policies require different reference monitor interfaces (i.e., different LSM hook placements) [10] (in the context of security kernels, see Chapter 6), the LSM uses the same placements for all LSMs. In practice, the choice of LSM hook placements was a union of the reference monitors being ported to the framework. The LSM implementation does not require that each LSM provide implementations for every hook, so the union approach does not demand extra work for LSM developers. Nonetheless, the set of LSM hooks has largely stabilized.

An example of the type of policy implemented by an LSM is the confinement policy of AppArmor [228]. AppArmor is a mandatory access control (MAC) system where the threat model is focused on the Internet. If we assume that systems are configured correctly, then the Internet is the only way that malicious input can reach the system. One threat is that network-facing daemons (e.g., `inetd`) are susceptible to malicious inputs (e.g., buffer overflows, format string vulnerabilities, etc.). AppArmor uses confinement policies for such network-facing daemons, that have traditionally been run with full privilege (e.g., `root`), to prevent compromised daemons from compromising the entire system.

9.2 SECURITY-ENHANCED LINUX

Security-Enhanced Linux (SELinux) is a system for enforcing mandatory access control that is based on the LSM framework [195, 229]. As shown in Figure 9.2, SELinux consists of a Linux Security Module and a set of trusted services for administration and secure system execution. In this section, we detail the SELinux reference monitor (Sections 9.2.1–9.2.4), then we discuss trusted services for administration (Section 9.2.5) and general trusted services (Section 9.2.6). We conclude this section with an evaluation of the SELinux system against a secure operating system specification in Definition 2.5.

The SELinux reference monitor consists of an authorization module and policy store. The SELinux authorization module builds authorization queries for a mandatory protection system (see Definition 2.4) in the SELinux policy store. SELinux uses fine-grained and flexible models for its protection state, labeling state, and transition state that cover all Linux system resources that are considered security-sensitive. Thus, the SELinux mandatory protection system enables comprehensive control of all processes, so policy writers can exactly define the required accesses. However, the low-level nature of the policy models results in complex policies that are difficult to relate to secrecy and integrity goals (e.g., information flow goals of Chapter 5). Nonetheless, the SELinux approach accurately demonstrates the challenges we face in ensuring that a commercial system enforces intended security goals.

9.2.1 SELINUX REFERENCE MONITOR

The SELinux reference monitor consists of two distinct processing steps. First, the SELinux reference monitor converts the input values from the LSM hooks into one or more authorization queries. These LSM hooks include references to Linux objects (e.g., file and socket object references), and

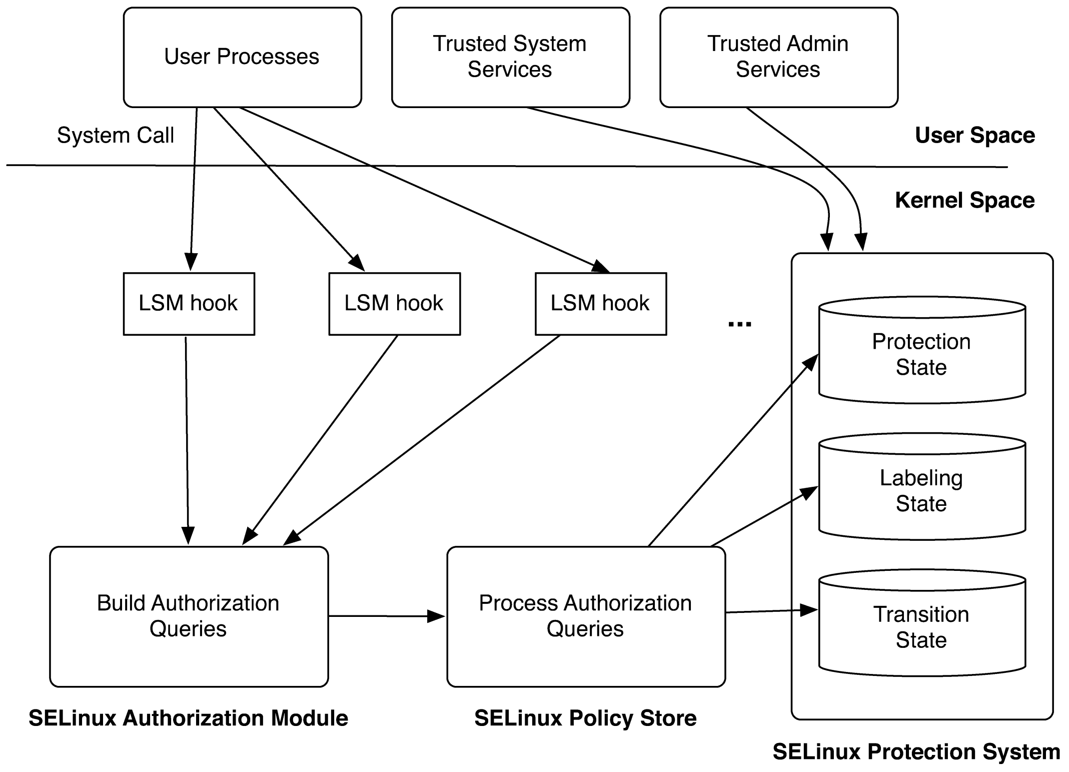

Figure 9.2: SELinux System Architecture

in some cases, argument flags, but the SELinux reference monitor must convert these into SELinux authorization queries (see below). Second, the core of the SELinux reference monitor processes these authorization queries against the SELinux protection system (i.e., the protection state, labeling state, and transition state). The SELinux protection system representation is highly optimized to support fine-grained authorization queries efficiently.

Consider the example below, the SELinux implementation behind the LSM hook that authorizes a file open system call.

```
static int selinux_inode_permission(struct inode *inode, int mask,
                                     struct nameidata *nd)
{
        if (!mask) {
                /* No permission to check. Existence test. */
                return 0;
        }
        return inode_has_perm(current, inode,
                                file_mask_to_av(inode->i_mode, mask), NULL);
}
```

Recall that when `open` system call is invoked, the target file is specified by a UNIX pathname and the requested operations are specified using a bit vector `flags` [2]. The kernel's implementation of `open` resolves the pathname down to the actual `inode` that refers to the target file, and then it invokes the LSM hook to authorize whether the requesting process can perform the requested operations on the resultant `inode`.

The function `selinux_inode_permission` above has three arguments, the `inode` for the file, the `mask` that indicates the file operations, and `namedata` related to the file path (not used in this authorization).

The SELinux implementation identifies the specific Linux objects corresponding to the subject, object, and operations for the authorization query. First, the subject of an `open` system call is the process that submitted the system call. In Linux, the process that invoked a system call is identified by the global variable `current`. As a result, this need not be an input from the LSM hook. Second, the object of an `open` call is the target `inode`. A reference to the `inode` is included in the LSM hook. Third, the operations requested by an `open` system call (e.g., read, write, and append) are determined from the `flags` input is sent to `selinux_inode_permission` function via the `mask` variable. The `namedata` is not used by the SELinux LSM, but may be used by other LSMs.

The subject (`current`), object (`inode`), and operations (results of the `file_mask_to_av`) are submitted to the function `inode_has_perm`, which generates the actual SELinux authorization query as shown below.

```
static int inode_has_perm(struct task_struct *tsk,
                           struct inode *inode,
                           u32 perms,
                           struct avc_audit_data *adp)
 {
        struct task_security_struct *tsec;
        struct inode_security_struct *isec;
        struct avc_audit_data ad;

        tsec = tsk->security;
        isec = inode->i_security;

        if (!adp) {
                adp = &ad;
                AVC_AUDIT_DATA_INIT(&ad, FS);
                ad.u.fs.inode = inode;
        }
```

[2]The `open` system call has a third argument `mode`, but that is not pertinent to this example. Its implications are authorized elsewhere.

```
        return avc_has_perm(tsec->sid, isec->sid, isec->sclass, perms, adp);
}
```

Rather than submitting the objects directly in an authorization query, SELinux assigns labels to subjects and objects, called *contexts* in SELinux. As we describe in the next section, *subject contexts* define a set of permissions (objects and operations) available to processes running with that context. An *object context* groups a set of objects that have the same security requirements. As required for a mandatory protection system, the set of contexts must be fixed, so the protection state is immutable. Likewise, the labeling and transition states in an SELinux system are immutable as well.

In SELinux, the kernel stores a context with each process and system resource that may appear in an LSM hook. For processes, its data type `task_struct` includes the field `security` in which the subject context of the process is stored. For the `inode` data type, a field `i_security` stores its object context. The function `inode_has_perm` extracts the subject and object contexts for these input arguments (i.e., `tsk` and `inode` in `inode_has_perm`) and generates the SELinux authorization query, defined by the function `avc_has_perm`. This function takes four arguments: (1) the subject context; (2) the object context; (3) the SELinux classification for the object's data type; and (4) the operations requested in the query. The SELinux policy store executes this query, determining whether the subject context can perform the requested operations on an object with the specified object context and SELinux classification for its data type. Such classifications correspond to system data types, such as `file` and `socket`, as well as more specific subtypes of these.

SELinux defines authorization queries for nearly all of the LSM hooks. For most of these LSM hooks, a single authorization query is generated, but in some cases, multiple authorization queries are generated and evaluated. For example, in order to send a packet, the process must have access to send using the specified port, network interface, and IP address (see `selinux_sock_rcv_skb_compat` for an example). Each of these authorization queries must be authorized for the operation to be permitted.

Authorization queries on protection state retrieve the SELinux policy entry that corresponds to the subject context, object context and object data type. The policy entry contains the authorized operations for this combination, and `avc_has_perm` determines whether all the requested operations are permitted by the entry. If so, the operations are authorized and the SELinux implementation returns 0. The Linux kernel is then allowed to execute the remainder of the system call (or at least until the next LSM hook).

9.2.2 SELINUX PROTECTION STATE

The SELinux reference monitor enforces the SELinux protection state, labeling state, and transition state. First, we discuss the SELinux protection state. SELinux contexts described above represent the SELinux protection state. They are a rich representation of access control policy enabling the definition of fine-grained policies. In this section, we define the concepts in an SELinux context and describe how they express security requirements over Linux processes and system resources.

SELinux Contexts Figure 9.3 shows the concepts that define an SELinux context and their relationships. A *user* is the SELinux concept that comes closest to a UNIX user identity (UID). The

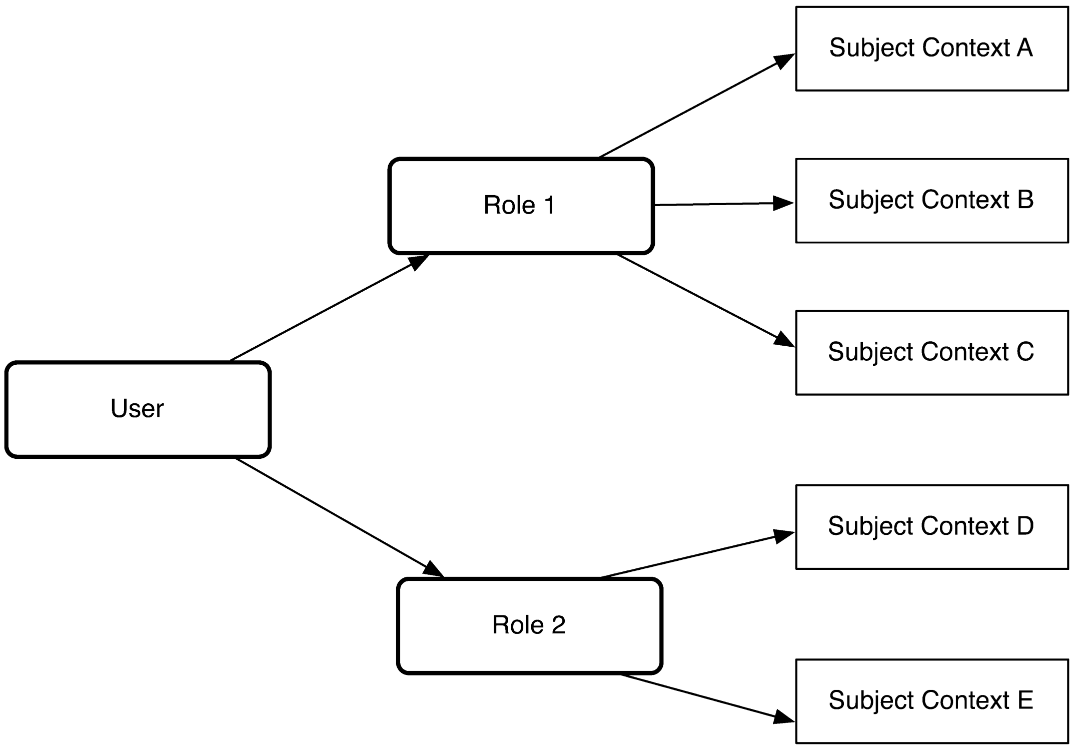

Figure 9.3: SELinux Contexts: *user* limits the set of *roles* that can be assumed to *Role 1* or *Role 2* (only one role). *Roles* limit the set of *subject types* and the *MLS range* that a process can assume (and hence, the permission available to the user). A context may have only one subject type, but a process can transition among all the subject types associated with a role (if authorized).

user is the authenticated identity of a user in the system. This typically corresponds to UNIX user identity in name, but does not convey any access rights to the user. In SELinux, user identity solely defines a set of roles to which the user is permitted by the SELinux policy. When a user authenticates (e.g., logs in), the user can choose one role from the set authorized by the SELinux policy.

An SELinux *role* is similar to the concept of a role in a role-based access control (RBAC) model [94, 268, 272] in that it limits the set of permissions that the user may access. However, unlike an RBAC role, the role is not assigned permissions directly. Rather, an SELinux role is associated with a set of *type* labels, as in a Type Enforcement (TE) model [33], and these type labels are assigned permissions. The role also optionally determines the *MLS range* that processes in that role may assume. When a user is authenticated, this determines the user's role and all the processes that the user runs must have a type label and MLS range that is authorized for the user's role.

For example, a user authenticates to the identity `alice` under the user role `user_r`. This role is permitted to run typical user processes under the type label `user_t`, but is also allowed to run other processes with type labels that `user_r` is authorized for. The user role `user_r` permits the user `alice` to use the `passwd` program to change her password, but since this program has different permissions than a typical user program (e.g., it can modify `/etc/shadow`, the file containing the hashed user passwords) it runs under a different type label, `passwd_t`. As a result, the user role `user_r` is authorized to run processes under the `user_t` and `passwd_t` type labels.

Type labels are assigned to permissions using `allow` statements as shown below.

```
allow <subject_type> <object_type>:<object_class> <operation_set>
allow user_t         passwd_exec_t:file           execute
allow passwd_t       shadow_t:file                {read write}
```

An `allow` statement associates a subject type (e.g., `user_t`) with permissions described in terms of an object type (e.g., `passwd_exec_t`, the label of the `passwd` executable file), the data type of the object (e.g., the `passwd` executable is a file), and the set of operations on the object type authorized by the `allow` statement (e.g., `execute`). Thus, the first `allow` statement permits any process with the type label `user_t` to execute any file labeled with the `passwd_exec_t` type label. The second `allow` statement permits any process running with the `passwd_t` type label to read or write any file with the `shadow_t` type label. This limits access to `/etc/shadow` to only those processes that run under the `passwd_t` type label, typically only processes running `passwd`.

The SELinux MLS labels represent a traditional lattice policy consisting of *sensitivity levels* and *category sets*. The MLS labels are interpreted according to the Bell-LaPadula model [23] (see Chapter 5) for read operations, but a more conservative interpretation is used to authorize write operations. For a read operation, the subject's sensitivity level must dominate (i.e., be greater than) or be equal to the object's sensitivity level, and the subject's category sets must include all (i.e., be a superset of) the object's category sets (i.e., the *simple-security property*). For a write operation, the subject's sensitivity level must be equal to the object's sensitivity level, and the subjects category sets must equal those of the object as well. This is more restrictive than the usual MLS write rule based on the ⋆-security property. The ⋆-security property permits write-up, the ability for subjects in lower sensitivity levels to write to objects in higher sensitivity levels. The ⋆-property presumes that the lower secrecy processes do not know anything about the high secrecy files (i.e., the subject's sensitivity level is dominated by the object's), such as whether a higher secrecy file of a particular name exists. However, Linux processes are not so unpredictable, so it may be possible for one Linux process to guess the name of a file used by a higher secrecy process, thus impacting the integrity of the system. As a result, write-up is not permitted in SELinux.

The SELinux reference monitor authorizes whether the subject's type label and MLS label both permit the requested operations to be performed on the request object based on its type label and MLS label. The two labels are authorized independently as described above. For type labels, an `allow` rule must be defined that permits the subject type to perform the requested operations on the corresponding object type. In addition, the MLS labels of the subject and object must also permit the requested operation. Both authorization tests must pass before the operation is authorized.

SELinux Policies The SELinux protection state enables comprehensive, fine-grained expression of a system's security requirements. First, each distinct object data type and operation in a Linux system is distinguished in the SELinux protection state. The SELinux protection state covers all security-sensitive system resource data types, including various types of files, various types of sockets, shared memory, interprocess communications, semaphores, etc. There are over 20 different object data types in all. In addition, SELinux provides a rich set of operations for each data type. In addition to the traditional `read`, `write`, and `execute` operations on a file, the standard file data type in SELinux has operations for `create`, `ioctl`, `fcntl`, extended attributes, etc. As a result, comprehensive and fine-grained control of system resources is possible.

Second, each process and object with different security requirements requires a distinct security context. If two processes cannot access exactly the same set of objects with exactly the same set of objects, then two distinct subject type labels are necessary, one for each process. Then, the appropriate `allow` statements for each can be defined. For example, distinct subject types for `user_t` and `passwd_t` had to be defined because `passwd` can access the `/etc/shadow` whereas a typical user process cannot. Further, if two objects cannot be accessed by exactly the same processes, then they also require two distinct object type labels. Again, the `shadow_t` and `passwd_exec_t` object type labels are necessary because these two files (`/etc/shadow` and the `passwd` executable) cannot be accessed by all processes using the same operations. As a result, over 1000 type labels are defined in the SELinux reference policy, the default SELinux protection state, and tens of thousands of `allow` statements are necessary to express all the different relationships between subjects and objects in a Linux system.

While the SELinux policy model results in complex protection state representations, the protection state complexity is a result of the complexity of Linux systems. Linux consists of many different programs, most with distinct access requirements and distinct security requirements, resulting in a large number of type labels. The large number of type labels then requires a large number of `allow` statements to express all the necessary access relationships. The SELinux reference policy demonstrates what we are up against in trying to build secure Linux systems.

9.2.3 SELINUX LABELING STATE

Since the SELinux protection state is defined in terms of labels, as is typical of a mandatory access policy, the protection state must be mapped to the actual system resources. We suspended some degree of disbelief in the last section because, while we mentioned that certain files, such as `/etc/shadow`, had certain labels, such as `shadow_t`, we did not specify how the files obtained these labels in the first place. Further, processes are also assigned labels, such as the `passwd` process having the `passwd_t` label, and the mapping of labels to processes must also be defined.

These specifications are provided in what we call the *labeling state* of the mandatory protection system in Definition 2.4. The labeling state is an immutable policy that defines how newly created processes and system resources are labeled. SELinux provides four ways in which an object's label can be defined.

First, an object may be labeled based on its location in the file system. Suppose the files `/etc/passwd` and `/etc/shadow` are provided in a Linux package for the `passwd` program. In this case, the file already exists in some form and needs to be labeled when it is installed. SELinux uses *file contexts* to label existing files or files provided in packages. A file context specification maps a file path expression to an object context. The file path expression is a regular expression that describes a set of files whose file path matches that expression. Below, we list two file contexts specifications.

```
<file path expr>  <context>
/etc/shadow.*     system_u:object_r:shadow_t:s0
/etc/*.*          system_u:object_r:etc_t:s0
```

For example, the second file context specification above defines the object context for files in the `/etc` directory. `/etc/shadow` gets a special context while other files in `/etc` (e.g., `/etc/passwd`) get the default context [3].

Second, for dynamically created objects, their labels are inherited from their parent object. For files, this is determined by the parent directory. For all files dynamically created in the `/etc` directory, they inherit the label defined for the directory, `etc_t`.

Third, `type_transition` rules can be specified in the SELinux policy that override the default object labeling. Below, we show a `type_transition` rule that relabels all files created by processes with the `passwd_t` type that would be assigned the `etc_t` label by default to the `passwd_t` label.

```
type_transition <creator_type> <default_type>:<class> <resultant_type>
type_transition passwd_t       etc_t:file             shadow_t
```

Note that the creating process context must be authorized to relabel these `etc_t` files to `passwd_t` files [4]. If we use the `passwd` process to create `/etc/shadow`, where `/etc` has the `etc_t` label, it would be assigned a `shadow_t` label instead based on this rule.

The SELinux labeling state enforces security goals through the administrator-specified file contexts, default labeling, and authorized `type_transition` rules. The labeling state enables precise control over labeling, but does not necessarily ensure a coherent security goal (i.e., information flow). An external analysis is necessary to determine whether labeling state achieves the desired security, as we discuss in the SELinux evaluation.

[3]Note that the user and role for all SELinux objects are `system_u:object_r:`.

[4]In order to relabel an object from type `T1` to type `T2`, the subject must have `allow` rules that permit `relabelfrom T1` and `relabelto T2`.

9.2.4 SELINUX TRANSITION STATE

By default, a process is labeled with the label of its parent, as described above, but the SELinux transition state enables process label transitions. If a user shell process runs with the `user_t` label, then all the processes that are created from this shell are also run under the `user_t` label. While this makes sense for many programs, such as editors, email client, and web browsers, some programs that the user runs need different permissions. For example, the `passwd` program needs access that must not be permitted for typical user programs, such as write access to `/etc/passwd` and read-write access to `/etc/shadow`.

SELinux `type_transition` rules are also used to express such process label transitions. As shown below the syntax is similar to the object labeling case, but the semantics are slightly different.

```
type_transition <current_type> <executable_file_type>:process <resultant_type>
type_transition user_t       passwd_exec_t:process            passwd_t
```

For process label transitions, a `type_transition` rule specifies that a process running in a specific label (i.e., the `current_type`) executes a file with a specific label (i.e., the `executable_file_type`), then the process is relabeled to the `resultant_type`.

As is the case for object labeling, process label transitions on execution must be authorized. This requires three SELinux permissions: (1) the process must have `execute` access to the executable file's type; (2) the process must be authorized to transition when executing that file; and (3) the process must be authorized to transition its label to the `resultant_type`. In the case above, the user shell forks itself and executes the `passwd` file. At execution time, the `type_transition` rule is invoked. The SELinux reference monitor retrieves such rules, and authorizes the conditions necessary to invoke the rule. If the transition is authorized, then the process is run using the `passwd_t` label, and it is able to access the `/etc/passwd` and `/etc/shadow` files as necessary.

Note that SELinux process label transitions are only permitted at process execution [5]. When a process is executed, the old process image is replaced with a new image defined by the file being executed, so the process context can be reassigned based on this image. Note that there may be some carryover from the old process, such as the set of file descriptors that are left open on execute and the processes environment variables, but the SELinux transition rules can limit the contexts under which transitions are allowed. For example, if a program depends on being run with a high integrity set of environment variables, then only transitions from high integrity contexts should be permitted. In the case of `passwd`, it is run from untrusted user processes, so the `passwd` executable must be trusted to protect itself from any low integrity inputs provided at execute time.

SELinux process transitions are more secure than traditional UNIX process transitions via `setuid` in several ways. First, a `setuid` transition almost always results in a process running with full system privileges (i.e., a `setuid root` process). In SELinux, the process transitions to a specific label with limited permissions defined for its purpose. Second, UNIX permits any process to execute a `setuid` program. As a result, all `setuid` programs are susceptible to malicious input from untrusted

[5] SELinux now has a command that permits process context transitions at any time, called `setcon`, but this command must be used carefully to prevent a process from obtaining unauthorized permissions. In general, use of this command is not recommended.

invocations. In SELinux, the contexts under which a process may be invoked can be limited. For example, SELinux rules can be written to ensure that only trusted contexts can execute a program and obtain all its rights.

SELinux transitions are comparable to Multics transitions, see Chapter 3. In Multics, ring brackets limit which processes may cause a process label transition by executing more trusted code. However, SELinux controls are finer-grained, as they can be defined at the level of an individual program, rather than a protection ring. However, Multics defines a formal concept for ensuring that a protection domain is protected from malicious inputs, the *gatekeepers*. SELinux has no such concept, but depends on the program developer to ensure protection.

Finally, SELinux now provides mechanisms that enable a process to relabel itself or system resources using the `setcon` and `chsid` commands, respectively. For system resources, the `passwd` process can explicitly invoke `chsid` to relabel `/etc/shadow` to the `shadow_t` label. Any process that is SELinux-aware can request a file relabeling, but the SELinux reference monitor authorizes all these transitions. That is, a `passwd_t` must also be authorized to relabel `passwd_t` files to `shadow_t`.

9.2.5 SELINUX ADMINISTRATION

As SELinux uses a mandatory access control (MAC) policy, only system administrators may modify its protection system's states. As a result, these states are generally static. SELinux provides two mechanisms for updating its protection system: (1) monolithic policy loading and (2) modular policy loading. In either case, configuring SELinux policies is a task for experts, so only a small number of policies have been developed.

Monolithic Policies The traditional SELinux protection system states are defined as a single, comprehensive, binary representation generated from the policy statements (e.g., `allow`, `type_transition`, etc.) described above. The SELinux policy compiler `checkpolicy` builds such policy binaries. For a monolithic policy, the tens of thousands of SELinux policy statements are compiled into a binary that is over 3 MBs in size.

The trusted program `load_policy` enables an administrator to load a new protection state that entirely replaces the old protection state. `load_policy` uses the Linux Sysfs file system to load the binary into the Linux kernel where the SELinux reference monitor in the kernel can use it. All authorization queries are checked against the policy binary.

Modular Policies As the SELinux policy is actually defined per Linux program and Linux programs themselves may be installed incrementally via packages, the SELinux policy administration mechanisms have also been extended to support incremental modification. SELinux policy modules define program-specific protection state contributions. A comprehensive SELinux policy binary is constructed from these individual modules. A policy module consists of four parts: (1) its own type labels and `allow` rules for these types; (2) its file context specification defining how its files are labeled; (3) its interfaces that enable other modules to access its type labels; and (4) this module's use of other module's interfaces.

The definition of type labels, `allow` rules, and file contexts are no different than for the monolithic policy, described in examples above, but the policy modules add the concept of module interfaces [324]. Module interfaces, like public method interfaces in object-oriented programs, provide entry points for other modules to access a module's type labels. For example, an interface definition specifies a set of `allow` rules that are permitted to the module invoking the interface. For example, the kernel policy module defines an interface `kernel_read_system_state(arg)` where the type label submitted as the argument `arg` is assigned to allow rules that permit read access to system state. A policy module specifies both its own interfaces and the set of interfaces that it uses. The function `semodule` is used to load new modules into the SELinux policy binary.

Policy Development Originally, two types of SELinux policies were developed: (1) a strict policy and (2) a targeted policy. The strict policy aims to enforce least privilege over all Linux programs, thus maximizing the protection possible while permitting reasonable functionality. The strict policy presents two challenges to deployment. First, it may be more restrictive than the Linux programs expect, leading to the failure of some programs to run properly. Second, the strict policy does not enforce any formal secrecy or integrity goal, so the policy may still permit significant vulnerabilities.

The targeted policy principle was introduced by the AppArmor LSM [228], and it defines *least privilege* policies for network-facing daemons to protect the system from untrusted network input. Other programs run without restriction. This limits the task of configuring restrictive policies to just the network-facing daemons, which simplifies policy expression and debugging. However, the targeted policy does not protect the system from other low integrity inputs (e.g., malicious emails, downloaded code, malware that is installed under a different label). As a result, the targeted policy is more appropriate for server systems whose software is carefully managed, but which may be susceptible to malicious network requests. In practice, SELinux distributions (e.g., RedHat) are delivered with a *targeted* SELinux policy.

Recently, a third SELinux policy, the reference policy, has been defined [309]. The reference policy enables an administrator to build either the targeted or strict policy from a single set of policy files. A configuration file enables administrators to describe their specifications for building policies. The reference policy also includes MLS support by default.

9.2.6 SELINUX TRUSTED PROGRAMS

In addition to the administrator operations above to load policy (i.e., `load_policy` and `semodule`), there are many other user-level programs that are trusted to specify and/or enforce the SELinux security requirements for the SELinux system to be secure. These programs include authentication programs (e.g., `sshd`) necessary to establish an authenticated user's subject context, system services necessary to bootstrap the system (e.g., `init`), and server programs which are depended upon to enforce the SELinux policy on their operations.

Authentication programs have been modified to understand SELinux contexts. When a user authenticates, these programs inform the SELinux reference monitor, so it can assign the proper subject context for that user's processes.

System bootstrap services are mainly trusted because they have broad permissions that may enable them to compromise the integrity of the SELinux reference monitor and/or policy. These services run with near full privilege and are trusted not to modify or circumvent policy. For example, such processes use the traditional UNIX fork/exec when they start system services (e.g., `vsftpd`), so that these obtain the proper set of access rights through process labeling (i.e., via `type_transition` rules), as described above.

SELinux also includes some server programs that have been modified to enforce SELinux policies. An example server is the SELinux X server [325]. The X server provides mechanisms that could enable one client to obtain secret information or compromise the integrity of another. This has long been known as a problem [85], and several implementations of access enforcement for windowing systems have been developed over the years [90, 86, 42, 199, 289, 95]. The SELinux community built a reference monitor interface for the X server, and defined a user-level policy server that can respond to authorization queries [43, 308]. The policy server design is general in that it can support authorization requests from multiple user-level processes, similarly to the Flask object managers (see Chapter 7). The aim is that the user-level policy could be verified to ensure that such trusted servers enforce a policy that is compliant with the SELinux system policy.

The SELinux MLS policy contains over 30 subject types that are trusted by the system. In many cases, the subject types are associated one-to-one with programs, but some subject types, such as `init`, have many scripts that are run under a single trusted type. The larger the amount of trusted code, the more difficult it is to verify tamperproofing and verify correctness.

We note here that there are certain programs that SELinux does not trust. For example, SELinux does not trust NFS [267] to return a file securely. As a result, SELinux associates an `nfs_t` type label for all these files, regardless of their label on the NFS server. The reason for this is that the NFS server delivers files to its clients in the clear, so an attacker may reply with a false file upon an NFS request. File systems with integrity-protected communication, such as kerberized Andrew File System [225, 234], could potentially be trusted to deliver labeled files. A variety of distributed files systems that provide secure access to files have been designed [28, 2, 197, 104, 255, 314]. A detailed treatment of this subject is beyond the scope of this book.

9.2.7 SELINUX SECURITY EVALUATION

We now assess whether SELinux satisfies the secure operating system requirements of Chapter 2. SELinux provides a framework in which these requirements can be satisfied (i.e., it is "secureable" like Multics), but the complexity of UNIX-based systems makes it difficult to provide complete assurance that these requirements have been met. Further, the practical requirements of UNIX systems (i.e., the function required) limits our ability to configure a system that would satisfy these requirements. As a result, SELinux provides significant security improvements over traditional UNIX systems (see Chapter 4), but it is difficult to quantify these improvements to the extent required of a secure operating system.

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated without creating security problems, such as TOCTTOU?

   The Linux Security Modules framework's reference monitor interface is designed to authorize access to the actual objects used by the kernel in security-sensitive operations to prevent vulnerabilities, such as TOCTTOU.

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

   The Linux Security Modules framework mediates operations identified by the LSM community to lead to security-sensitive operations. The mediation provided is effectively a union of all the Linux reference monitor prototype's constructed.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

   Since the LSM framework's reference monitor interface was designed in an informal manner, verification that it provides complete mediation is necessary. Source code analysis tools were developed to verify that the security-sensitive kernel data structures were mediated [351] in a consistent manner [149], and bugs in the LSM reference monitor interface were found and fixed. However, these tools are an approximation of the complete mediation requirements, and they are not applied on a regular basis. Nonetheless, no errors in the reference monitor interface placement have been found since its introduction in Linux 2.6.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, from modification?

   The LSM reference monitors, such as SELinux, are run in the supervisor protection ring, so they are as protected as the kernel. Although the LSM framework is a module interface, LSMs are compiled into the kernel, so they can be active at boot time.

   The Linux kernel can be accessed by system calls, special file systems, and device files, so access to these mechanisms must ensure tamper-protection. While Linux system call processing does not provide input filtering at the level of Multics *gates*, work has been done to verify kernel input handling using source code analysis tools [154]. Further, Linux systems provide a variety of other operations that enable access to kernel memory. For example, special file systems, such as the `/proc` filesystem and *sysfs* filesystem, and device files enable access to kernel memory through files. The SELinux protection state is configured to limit access to trusted processes (i.e., those with trusted subject type labels).

5. **Tamperproof**: Does the system's protection system protect the trusted computing base programs?

   As described above, SELinux system tamper-protection also requires that its trusted user-level programs be tamper-protected. An evaluation of SELinux policy showed that a set of trusted processes which defined a tamper-protected, trusted computing base could be identified [150]. However, several of these trusted processes must be trusted to protect themselves from some low integrity inputs, so satisfying a classical information flow integrity where no low integrity inputs are received (e.g., Biba integrity protection [27]) appears impractical. Some trusted SELinux services (e.g., `sshd` and `vsftpd`) were shown to enforce a weaker version of Clark-Wilson integrity [54, 285].

6. **Verifiable**: What is basis for the correctness of the system's trusted computing base?

   As is typical, verifying the correctness of security enforcement is the most difficult task to achieve. Verifying correctness of the implementation of the Linux kernel and trusted programs is a very complex task. For this large a code base, written mostly in nontype safe languages, by a variety of developers, verification cannot be complete in practice. Linux has been assured at the Common Criteria evaluation level EAL4 (see Chapter 12), which requires documentation of the low-level design of the kernel. Converting this low-level design into a model in which security properties can be verified would be a challenging task, and it may be impractical to verify how the source code implements the design correctly.

7. **Verifiable**: Does the protection system enforce the system's security goals?

   SELinux policies define a precise, mandatory specification of the allowed operations in the system. As a result, it is possible to build an information flow representation from the SELinux policies [150] (mentioned above), even one that includes the transition state. Also, the MLS policy ensures information flow secrecy satisfies the simple-security and $\star$-security properties. However, the integrity analysis and MLS policy reveal a significant number of trusted subject types (over 30 for integrity and over 30 for MLS, and only some overlap). Thus, the SELinux approach enables the system to be "secureable," but system developers will need to manage the use of trusted code carefully to ensure the verified security goals are really met.

9.3 SUMMARY

The LSM/SELinux system implements a reference monitor in the Linux operating system. The LSM community emerged from a variety of prototype efforts to add a reference monitor to Linux, and developed a reference monitor interface that was acceptable to the security community (for the most part) and to the mainstream Linux community. The SELinux and AppArmor LSMs have been adopted by the major Linux distributors and are supported by many other distributors. While the resulting LSM framework has been only semi-formally tested, it has generally been a successful addition to the kernel. However, the combination of the Linux kernel and LSM framework is too complex for a complete formal verification that would be required to prove complete mediation and tamperproofing.

The challenge has been how to use the LSM reference monitor interface to enforce security goals. We examined the SELinux system which provides a comprehensive set of services for implementing security policies and a fine-grained and flexible protection system for precise control of all processes. The SELinux approaches demonstrates the complexity of UNIX systems and the difficulty in enforcing comprehensive security. The outstanding challenge is the definition and verification of desirable security goals in these low-level policies. The AppArmor LSM uses a targeted policy to protect the system from network malice, but proof of security goal enforcement will also have to verify requirements, such as information flow.