# Meeting Critical Security Objectives with Security-Enhanced Linux

Peter A. Loscocco, NSA, loscocco@tycho.nsa.gov
Stephen D. Smalley, NAI Labs, ssmalley@nai.com

## Abstract

Security-enhanced Linux incorporates a strong, flexible mandatory access control architecture into Linux. It provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements. This allows threats of tampering and bypassing of application security mechanisms to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications. Using the system's type enforcement and role-based access control abstractions, it is possible to configure the system to meet a wide range of security needs. This paper describes how Security-enhanced Linux was used to meet a number of general-purpose system security objectives.

## 1 Introduction

Operating system security is fundamental to the security of every computing system because operating systems are a critical point of failure for the entire system. Unfortunately, attempts to secure computer systems continue to be based on the flawed assumption that adequate security can be provided in applications with the existing security mechanisms of mainstream operating systems. The reality is that secure applications require secure operating systems, and any effort to provide system security which ignores this premise is doomed to fail. The integration of *Mandatory Access Control* (MAC) is a necessary step in the complex task of building a completely secure operating system. It would significantly improve system security and enable protection from many vulnerabilities that plague systems today [14].

A general purpose MAC architecture needs the ability to enforce an administratively-set security policy over all subjects and objects in the system, basing decisions on labels containing a variety of security-relevant information. When properly implemented, it enables a system to adequately defend itself and offers critical support for application security by protecting against the tampering with, and bypassing of, secured applications. It allows critical processing pipelines to be established and guaranteed. MAC provides strong separation of applications that permits the safe execution of untrustworthy applications. Its ability to limit the privileges associated with executing processes limits the scope of potential damage that can result from the exploitation of vulnerabilities in applications and system services. MAC enables information to be protected from legitimate users with limited authorization as well as from authorized users who have unwittingly executed malicious applications. The ability for the system to do these types of things is necessary before the construction of secure systems will be possible.

It is impossible to obtain the benefits derived from MAC with existing *Discretionary Access Controls* (DAC) like those currently found in Unix systems. It is not adequate to base access decisions only on user identity and ownership. It must be possible to consider additional security-relevant criteria such as the role of the user, the function and trustworthiness of programs, or the sensitivity or integrity of the data. As long as users have complete discretion over objects, it will not be possible to control data flows or enforce a system-wide security policy.

Protection against malicious code is not possible using existing DAC mechanisms because every program executed by the user inherits all of the privileges associated with that user. Malicious programs are free to change the permissions associated with all of the user's objects, as well as disclose or alter the objects themselves. This problem is exacerbated by the fact that only two categories of users are supported, completely trusted administrators and completely untrusted ordinary users. Many system services and privileged programs must run with coarse-grained privileges that far exceed their requirements. A flaw in any one of these programs can be exploited to obtain complete system access.

Traditional MAC mechanisms have typically been too limiting to serve as a general security solution. They have tended to be too tightly coupled to a multi-level security (MLS) [5] policy which bases its access decisions on clearances for subjects and classifications for objects. This traditional approach is too limiting to meet many security requirements [6, 7, 8]. It provides poor support for data and application integrity, separation of duty, and least privilege requirements. It requires special trusted subjects that act outside of the access control model. It fails to tightly control the relationship between a subject and the code it executes. Thus, traditional MAC systems have limited ability to offer protection based on the function and trustworthiness of the code, to correctly manage permissions required for execution, and to minimize the likelihood of malicious code execution.

The *Flask* Architecture [17] was created as an attempt to serve as a general architecture for MAC. An important design goal was to provide flexible support for security policies, since no single MAC policy model is likely to satisfy everyone's security requirements. This goal was achieved by cleanly separating the security policy logic from the enforcement mechanism. Care was taken to ensure that well-defined policy interfaces were specified that could support the widest set of useful security policies. The architecture provides support for policy changes and allows security policies to be expressed naturally in terms that make sense to the particular security policy that is implemented. In addition, with the Flask architecture, the enforcement of the security policy can be transparent to applications because it is possible to define default security behavior.

*Security-Enhanced Linux* [13, 12], or *SELinux* for short, is an application of the Flask architecture in the Linux operating system. MAC has been integrated into the major subsystems of the Linux kernel, including fine-grained controls for operations on processes, files, and sockets. The security policy decision logic has been encapsulated into a new kernel component called the *Security Server* (SS) which makes labeling, access and polyinstantiation decisions in response to policy-independent requests that have been placed throughout the kernel. This architecture enables the kernel to enforce policy decisions without needing access to the details of the policy.

The SS implemented for SELinux was designed with a particular model of MAC selected to address the limitations of traditional MAC implementations. Because of the flexibility of the Flask architecture, this model is easily modified, or even replaced, to support other models as well. In general, this will not be necessary because the SS released with SELinux is capable of supporting many security policies that meet a wide variety of security objectives. It achieves significant policy flexibility using configuration files to define security policies that are easily tailored to meet specific installation needs.

The remainder of this paper is a discussion of how SELinux with its current SS implementation can be used to dramatically increase the level of security possible in a Linux system. It begins with a description of the model of security that the current SS implements. It is then shown how SELinux was used to meet some general security objectives. The paper ends with a short discussion of related work and some concluding remarks.

## 2 Security Server

The Flask architecture is sufficiently general to support many models of mandatory access control. A particular model must be chosen for every implementation of the security server. Once selected, a security server is constructed that makes all security relevant decisions in the context of that model. The security server is cleanly separated from the rest of the kernel, hidden behind a well-defined interface. This section describes the security server that was implemented for SELinux.

In choosing the security model for a security server implementation, care should be taken to ensure that it is sufficiently expressive to meet whatever security objectives are expected. The flexibility of the Flask architecture allows the security server to be modified, or even replaced, to alter the supported security model to meet additional requirements. The complete encapsulation of the security policy logic within the security server makes this possible without any impact on the rest of the system.

The content and format of labels used in the system depend on the particular security model implemented by the security server. Security decisions within the security server are based on *security contexts* which represent security labels. A security context is a policy independent data type that can be handled by different parts of the system but should only be interpreted by the security server. It contains all of the security attributes associated with a particular labeled object that are relevant to the policy decision logic.

Security contexts are usually not bound directly to objects. A second policy-independent data type called a *security identifier* (SID) is bound to each object that requires a label. SIDs are nonglobal and nonpersistent opaque objects that are mapped to security contexts. This mapping is created at run time and maintained by the security server. When an object is created, the security server decides which SID to use as a label. SIDs associated with objects are passed into the security server and used as the basis for security decisions.

The mandatory access controls of SELinux are implemented as permission checks that have been inserted at control points throughout the Linux kernel. Approximately 140 fine-grained permissions, grouped into 28 object classes, have been defined to allow the control of nearly every system operation. Examples of permissions are the *transition* and *signal* permissions in the *process* class or *create* and *write* permissions in the *file* class. Permission checks are made between a source SID and a target SID for a particular permission in some object class. Usually, but not always, these are the SIDs associated with a calling process and some object, like a file, that is being accessed. To respond to permission checks, the security server's policy logic uses the security relevant attributes contained in the security contexts associated with the source and target SIDs to determine if a permission can be granted.

### 2.1 Security Server Model of SELinux

The implementation of the example security server for SELinux required that a concrete security model be selected. A combination of *Identity-based Access Control* (IBAC), *Role-based Access Control* (RBAC), and *Type Enforcement* (TE) was chosen. As previously mentioned, SELinux does not depend on this model; it is straightforward to replace it with some other choice. It was chosen as the example because it is sufficiently general to support many security objectives found in real-world security.

The SELinux security server maintains security contexts with three relevant security attributes, an *identity*, a *role*, and a *type*. It determines which combinations of values for these attributes can be combined into security contexts. It uses each component of the security context to compute a portion of the access decisions.

There is an identity associated with every process on the system. Changes to it are rigorously controlled and the policy configuration only allows certain programs to change the identity (currently login and crond). When a user logs on and presents his credentials, the identity portion of the security context related to his login shell will reflect the user's real identity. SELinux identities are orthogonal to Linux UIDs. Except when specified in the policy, whenever the UID of a process is changed, its SELinux identity remains unchanged; when a new program is executed, the identity component will be preserved. In this way, all access control decisions can be based the correct identity.

The actions of any particular user are restricted by the RBAC policy. Users are assigned a set of roles that they may assume. The transition between roles is controlled by the policy. Although not strictly necessary, the security policy has been configured to limit role transitions to occur only as a result of running programs that require user authentication. This was done to ensure that roles changes can only occur with explicit user consent and not from executing some malicious program.

Roles are used to express allowable user actions. The RBAC policy used in SELinux differs from that described in [10] in that it defines allowable user actions for a particular role using the TE policy. Whereas a typical RBAC policy would directly specify permissions granted to roles, the SELinux RBAC policy specifies domains that can be entered by roles, and defers the assignment of permissions to the TE configuration. Although this form of RBAC does not offer additional security over a strict TE policy, it does allow the TE policy to be more easily managed using the higher-level concept of roles.

The TE policy is used to express the fine-grained access controls. Each object in the system is assigned a type. An access matrix is defined where each element of the matrix determines the allowable accesses between a pair of types. In SELinux, the accesses are expressed in terms of permissions granted for each subject and object class that were defined for the kernel control points. All permissions must be explicitly granted.

This form of TE differs from that defined in [6] in that there is no distinction made between types and domains. Domains are treated just as any other type. Domains are simply types assigned to processes. Because of this, types used as domains can also be used as a type of a related object, *e.g.* the type of its *procfs* entries. The term domain is often still used for convenience even though the security server does not internally distinguish them from types. By reducing domains and types to a single type abstraction, a single table can be used to express the TE policy based on type pairs rather than separate tables for subject interactions and object accesses. This also allows inter-object relationships to be defined in the policy.

This form of TE is also distinct in that the permissions are grouped by object class to facilitate expressing a matrix where so many more permissions are defined than typically is the case. The class concept allows permissions to be defined for each kind of object based on the services for that object, and it allows the policy to distinguish different kinds of objects, *e.g.* granting different permissions to a device file than to a regular file or to a raw socket than to a TCP socket. This enables SELinux to provide finer-grained permissions than what is typically expected when using TE.

TE offers many benefits over the traditional approach to MAC. The TE access matrix provides a clean separation of the policy and enforcement mechanisms. It is capable of supporting many policies. It is useful for expressing integrity, separation, containment, and invocation policies. No trusted subjects that can violate the policy are necessary. Because every process can be assigned a domain, every process can be controlled using exactly the same mechanism. Fine-grained permissions can be assigned to programs limiting privileged programs to the minimal permissions required to complete their task. Lastly, TE allows the relationship between a subject and its executable to be tightly controlled, enabling protection based on the function and trustworthiness of code and offering protection against the execution of malicious code.

## 2.2 Configuring Security Policies

The specific policy that is enforced by the kernel is dictated by security policy configuration files. These text-based configuration files allow the security server implementation to support many security policies. They account for a significant part of security policy flexibility possible with SELinux. Once a security policy specification is completed, it is straightforward to customize that policy to meet the specific needs of different installations. It is possible to maintain different security policies that address completely different security objectives with just one base system.

The configuration files are written using a simple language developed for the security server. The configuration is checked and compiled into a binary representation that is loaded into the security server at boot time. It may also be reloaded at runtime as controlled by the policy.

The TE configuration file defines an extensible set of types. Using the *allow* statement, allowable permissions between pairs of types are specified for each object class.

```
allow type_1 type_2:class { perm_1 ... perm_n };
```

The TE configuration file defines automatic transitions between types when programs of certain types are executed. Such transitions ensure that system processes and certain programs are placed into their own separate domains automatically when executed. It also defines default labels for files created by programs of certain types in certain types of directories to ensure that files are created with the right types. Both are done using the *type_transition* statement.

```
type_transition type_1 type_2: file
     default_file_type;

type_transition type_1 type_2: process
     default_process_domain;
```

The RBAC configuration file defines an extensible set of roles. Each process has an associated role.

The configuration file specifies the set of types that may be entered by processes executing in a given role. The *role* statement is used to define the roles.

```
role rolename types { type_1 ... type_n };
```

As users execute programs, transitions to other roles may, according to the policy configuration, automatically occur to support changes in privilege. 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. The *role_transition* statement is used to define the roles.

```
role_transition current_role program_type new_role;
```

Although the language allows role transitions to occur on program execution, the SELinux configuration never uses this functionality. Instead it uses domain transitions for changes in privileges during a session. Roles are only allowed to change on login or by executing the *newrole* program which causes a user authentication. Unlike the TE policy, the RBAC policy has no entrypoint controls to control the transition into roles. Care must be taken when granting this capability. Role changes tend to involve significant changes in privileges (*e.g* user becoming system administrator) whereas domain transitions tend to be finer-grained changes.

The IBAC configuration file defines each user recognized by the system security policy. Each user has a set of roles that may be entered by processes with the user's identity. This is specified with the *user* statement.

```
user username roles { role_1 ... role_n };
```

There are two other configuration files available to further specify a security policy. They are the *assert.te* and *constraints* files. The first allows the specification of TE assertions that must hold true for the expressed policy to be valid. The second uses boolean expressions to express restrictions on users or roles. Both are useful tools to construct security policies.

Lastly, the operation of SELinux depends not only on the security policy configuration but also on the labels of objects in the file system. New objects are labeled when they are created. When the object is moved onto persistent storage, a *persistent SID* (PSID) is stored with that object. The PSID represents a security context, and a mapping between them is stored within the file system. PSIDs are mapped into SIDs when the kernel accesses the object. See [13, 17] for a more complete discussion on PSIDS.

When existing file systems are brought into SELinux, labels must be assigned. The *file_contexts* configuration file specifies security contexts for files based on pathname regular expressions. The *setfiles* utility program reads this configuration and sets the security contexts on each file accordingly. When setfiles is run, the system stores the proper PSID with each file and updates the security context mapping. In this way, the security policy can depend on the file system labels.

## 3 Meeting Security Objectives

The SELinux release includes an example of a general-purpose security policy configuration designed to meet a number of security objectives as an example of how a system may be secured [16]. The example RBAC configuration is very simple. All system processes run in the *system_r* role. Two roles are currently defined for users, *user_r* for ordinary users and *sysadm_r* for system administrators.

Most of the policy is specified through the example TE configuration. Separate domains are defined for various system processes and authorized for the *system_r* role. Each user role has an associated initial login domain, the *user_t* domain for the *user_r* role and the *sysadm_t* domain for the *sysadm_r* role. This initial login domain is associated with the user's initial login shell. As the user executes programs, domain transitions occur automatically as needed to change privileges. Different sets of domains are authorized for each of the user roles.

The rest of this section describes how the TE configuration meets a specific set of security objectives. It provides and explains detailed examples of the configuration to address each objective. In some cases, macros in the actual configuration have been expanded for the excerpts in this section to reveal greater detail about the configuration. Additionally, in some cases, the full expansion of a macro has been pruned for brevity.

### 3.1 Limiting Raw Access to Data

Access controls for individual processes and files are of little use if an attacker can directly access raw data. Hence, the policy configuration must carefully limit raw access to data. The example configuration defines a set of types for objects that can be used to access raw data. Access to these types is only granted to a small set of privileged domains, and entry to these domains is carefully controlled.

Since `fsck` and related utilities must access the raw disk, a *fsadm_t* domain is defined for such utilities. A *fsadm_exec_t* type is assigned to the program files for these utilities. The following excerpt shows a portion of the configuration relevant to this domain:

```
allow fsadm_t fsadm_exec_t:process
      { entrypoint execute };
allow fsadm_t fixed_disk_device_t:blk_file
      { read write };
allow initrc_t fsadm_t:process transition;
allow sysadm_t fsadm_t:process transition;
```

The first statement in this excerpt allows the *fsadm_t* domain to be entered by executing a program labeled with the *fsadm_exec_t* type. The second statement allows the *fsadm_t* domain to read and write block special files with the *fixed_disk_device_t* type. The third statement allows the `rc` scripts to transition to this domain (e.g. so that `fsck` can be run automatically during initialization). The last statement allows an authorized system administrator to transition to this domain (e.g. so that `fsck` can be explicitly run by an administrator). Access to the raw disk device is controlled both with respect to the particular program and to the context in which the program is called.

Since `klogd` must access the kernel memory devices, a *klogd_t* domain is defined for this daemon. A *klogd_exec_t* type is assigned to the program file for the daemon. The following excerpt shows a portion of the configuration relevant to this domain:

```
allow klogd_t klogd_exec_t:process
      { entrypoint execute };
allow klogd_t memory_device_t:chr_file read;
allow initrc_t klogd_t:process transition;
```

This excerpt is very similar to the excerpt for *fsadm_t*. The *klogd_t* domain can only be entered by executing a program labeled with the *klogd_exec_t* type. The *klogd_t* domain can read character special files with the *memory_device_t* type. The `rc` scripts can transition to the *klogd_t* domain when the daemon is executed.

### 3.2 Protecting Kernel Integrity

A second goal of the example policy configuration is to prevent attackers from tampering with the kernel. An example of how the configuration protects the integrity of the kernel can be seen through its protections on the `/boot` files. Most of the `/boot` files are labeled with a *boot_t* type and can only be modified by an administrator. Since certain files in `/boot` are automatically updated during initialization, a separate *boot_runtime_t* type is defined for such files, and the domain for `rc` scripts is authorized to update these files. The following excerpt shows a portion of the configuration for these boot files:

```
allow initrc_t boot_t:dir
      { read search add_name remove_name };
allow initrc_t boot_runtime_t:file
      { create write unlink };
type_transition initrc_t boot_t:file boot_runtime_t;
```

The first statement allows the `rc` scripts to modify the `/boot` directory. The individual controls over each file ensure that this does not allow the `rc` scripts to remove or rename the existing files in `/boot` in order to replace them. The second statement allows the scripts to create or delete a file with the *boot_runtime_t* type. The last statement causes files created in `/boot` by the scripts to automatically default to the *boot_runtime_t* type.

A second example of how the policy configuration protects the integrity of the kernel can be seen in its handling of kernel modules. Distinct types are assigned to the module utilities and module object files to prevent unauthorized modification. The following excerpt shows a portion of the configuration for controlling the ability to insert kernel modules into a running kernel:

```
allow insmod_t insmod_exec_t:process
      { entrypoint execute };
allow insmod_t self:capability sys_module;
allow sysadm_t insmod_t:process transition;
```

The first statement allows the *insmod_t* domain to be entered by executing a program with the *insmod_exec_t* type. This type is assigned to the `insmod` utility. The second statement allows the *insmod_t* domain to use the *CAP_SYS_MODULE* capability to insert modules. The last statement allows system administrators to transition to this domain when they run the utility.

### 3.3 Protecting System File Integrity

Just as the integrity of the kernel must be protected, the integrity of other critical system files must also be protected. Separate types are defined and assigned to system software, system configuration information, and system logs to protect their integrity. The dynamic linker is labeled with the *ld_so_t* type. Many domains must be granted execute access to this type, since many programs are dynamically linked. System programs are labeled with types such as *bin_t* for ordinary programs or *sbin_t* for system administration programs. System shared libraries are labeled with the *shlib_t* type. Write access to these types is limited to administrators.

The protections applied to the `/etc` directory are an example of protecting system configuration files. Most files in `/etc` are labeled with the *etc_t* type and write access to this type is strictly limited. Since the `/etc/aliases` and `/etc/aliases.db` files and the `/etc/mail` directory must be modified by the `sendmail` program, separate types are defined for this file and directory, and the *sendmail_t* domain is granted write access to these types, as shown below:

```
allow sendmail_t etc_aliases_t:file
      { read write };
allow sendmail_t etc_mail_t:dir
      { read search add_name remove_name };
allow sendmail_t etc_mail_t:file
      { create read write unlink };
```

An example of protecting the integrity of system log files is the `wtmp` file, which stores login records. The policy configuration defines a *wtmp_t* type for this file. Separate domains are defined for programs (e.g. `login`, `utempter`, `gnome-pty-helper`) which must update this file, and write access is only granted for these domains. The following excerpt shows permissions granted to this type for several domains:

```
allow local_login_t wtmp_t:file { read write };
allow remote_login_t wtmp_t:file { read write };
allow utempter_t wtmp_t:file { read write };
```

### 3.4 Confining Privileged Processes

Flaws in privileged processes are often exploited to subvert the security of a system. The example configuration confines such processes by defining separate domains for them and restricting their accesses to least privilege. One example of such a process is `sendmail`. The following excerpt shows some of the permissions granted to the *sendmail_t* domain in which `sendmail` runs:

```
allow sendmail_t smtp_port_t:tcp_socket name_bind;
allow sendmail_t mail_spool_t:dir
      { read search add_name remove_name };
allow sendmail_t mail_spool_t:file
      { create read write unlink };
allow sendmail_t mqueue_spool_t:dir
      { read search add_name remove_name };
allow sendmail_t mqueue_spool_t:file
      { create read write unlink };
```

The first statement allows `sendmail` to bind to the SMTP port. The next two statements allow `sendmail` to manage the mail spool directory. The last two statements allow `sendmail` to manage the mail queue directory. Even if a flaw in `sendmail` is exploited, the set of accesses granted to the attacker is strictly limited to what is specified in the configuration.

Another example of a privileged process is `ftpd`. The following excerpt shows some of the permissions granted to the domain for this daemon:

```
allow ftpd_t wtmp_t:file append;
allow ftpd_t var_log_t:file append;
allow ftpd_t ls_exec_t:process execute;
```

The first statement allows `ftpd` to append to the `wtmp` file. The second statement allows `ftpd` to append to `/var/log/xferlog`. For even better protection, a separate type could be defined for this particular log file, e.g. *xferlog_t*, to strictly limit the daemon to that file. The last statement allows `ftpd` to execute the `ls` program. As with the `sendmail` example, an attacker who subverts `ftpd` can only perform the actions authorized by the configuration.

### 3.5 Separating Processes

To protect processes in one domain from interference by processes in another domain, the example policy configuration restricts process interactions. The ability to access the /proc entries for processes in other domains is only granted to certain privileged domains such as the domain for system administrators. This is shown by the following excerpt:

```
allow sysadm_t domain:dir { read search };
allow sysadm_t domain:{ file lnk_file } read;
```

The /proc entries for each process are labeled with the domain of the process. These two statements grant the *sysadm_t* domain permission to access the /proc entries for all domains. Most domains are only allowed to access the entries for processes in the same domain.

Similarly, the ability to trace other processes or send signals to other processes is typically limited to processes in the same domain, except for sending *SIGCHLD* to notify the parent of the completion of the child. The following excerpt shows a case where processes in different domains are allowed to send signals to each other:

```
allow user_t user_netscape_t:process
      { sigkill sigstop signal };
allow user_netscape_t user_t:process sigchld;
```

The first statement allows an ordinary user to send arbitrary signals to his netscape process. The second statement allows the netscape process to send SIGCHLD to the ordinary user process so that it can be reaped when it exits. Since there are no statements allowing an ordinary user process to send signals to an administrator process or to system processes, an ordinary user cannot interfere with these other processes. Likewise, since there are no statements allowing the user's browser to send any signal other than SIGCHLD to other user processes, malicious mobile code executed by the browser cannot kill the user's other processes.

Since many processes use temporary files, the configuration must ensure that processes in different domains cannot interfere with one another by accessing each other's temporary files. The configuration achieves this goal by defining a derived type for temporary files created by each domain and limiting access to that type to the corresponding domain. If a domain requires access to a temporary file created by another domain, then it can be granted permission to that type on a case-by-case basis. The following excerpt shows a portion of the configuration for the temporary file type defined for the ordinary user domain:

```
allow user_t tmp_t:dir
      { read search add_name remove_name };
allow user_t user_tmp_t:file
      { create read write unlink };
type_transition user_t tmp_t:file user_tmp_t;
```

The first statement authorizes the ordinary user domain to create and unlink files in the /tmp directory. The individual controls over each file ensure that the domain cannot unlink files created by other domains, e.g. the administrator domain. The second statement allows the ordinary user domain to create and access files with the *user_tmp_t* type. The last statement causes files created by the ordinary user domain in /tmp to be automatically labeled with this type. Similar statements are defined for the administrator domain and for other domains that use temporary files, with a separate type defined for each such domain. This ensures that ordinary users cannot create and use their own temporary files but does not allow them to access the temporary files of other domains.

The configuration likewise defines different types for the home directories and terminal devices used by the different domains for users. The following excerpt shows statements granting permissions to terminal devices and home directories for the ordinary user domain:

```
allow user_t user_tty_device_t:chr_file { read write };
allow user_t user_devpts_t:chr_file { read write };
type_transition user_t devpts_t:file user_devpts_t;
allow user_t user_home_t:file
      { create read write unlink };
```

### 3.6 Protecting the Administrator Domain

Since the administrator domain is highly privileged, the policy configuration must ensure that this domain can only be entered in a secure fashion. The example configuration only allows entry via domains for the `login` program and the `newrole` program. The `login` program is run in a separate domain for local logins than for remote logins so that the configuration can prohibit entry on remote logins, since such logins may bypass authentication via `.rhosts` files. However, users who are remotely logged in may still use the `newrole` program after login in order to enter this domain. The following excerpt shows some of the relevant statements:

```
type_transition getty_t login_exec_t:process
     local_login_t;
allow local_login_t sysadm_t:process transition;
allow newrole_t sysadm_t:process transition;
```

The first statement causes the `login` program to run in the *local_login_t* domain when it is executed by `getty`. A separate statement that is not shown causes the `login` program to run in a separate *remote_login_t* domain when it is executed by `rlogind`. The next statement allows the *local_login_t* domain to transition to the administrator domain. The last statement allows the `newrole` program to transition to the administrator domain.

As described in the previous subsection, the example configuration also protects the administrator domain by preventing processes in other domains from interfering with it. The administrator domain is also protected against the execution of malicious code by limiting it to executing approved types and by automatically transitioning unsafe software such as `netscape` to a more restricted domain.

## 4 Related Work

Two other security projects have developed flexible access control frameworks for the Linux kernel. The Rule Set Based Access Control (RSBAC) for Linux project [15] provides a general framework for kernel access control and a set of security policy modules. The Medusa DS9 project [3] provides a kernel access control architecture that allows processes and files to be placed into separate virtual spaces in accordance with a policy defined by a user-space authorization server.

Several other projects have developed particular access control mechanisms for the Linux kernel. The Domain and Type Enforcement (DTE) for Linux project [11] provides a variant of Type Enforcement that uses an implicit typing mechanisms based on pathnames. This project is based on the original DTE prototype [4], which also investigated how to configure the DTE controls to meet real security objectives [18]. SubDomain [9] provides a variant of DTE that is limited to confining programs and that directly specifies access control configurations in terms of programs and files rather than domains and types. The Linux Intrusion Detection System (LIDS) project [1] provides administratively-defined program-based access control lists for files along with a collection of other features. For further discussion on related work see [13].

Due to its highly flexible architecture and comprehensive controls, SELinux is capable of representing many of the security policies and mechanisms provided by these other projects. However, since SELinux was only designed to address mandatory access controls based on the labels of subjects and objects, it cannot directly represent some of the requirements of these projects. The Linux Security Module project [2] has been created to develop a common set of kernel hooks that can support the needs of all of the Linux security projects, with the goal of integrating this general set of hooks into the mainstream Linux kernel.

## 5 Conclusions

The integration of mandatory access controls into Linux is necessary in order to allow secure systems to be built with Linux. Since no single security model is suitable for all purposes, a general-purpose solution must be sought. SELinux has many of the properties that should be considered for a general-purpose security architecture.

SELinux is a comprehensive and flexible system with a well-defined MAC architecture that has been validated through several prototypes. It cleanly separates policy decisions from their enforcement using a general interface. It provides support for policy changes and is independent of policy, policy languages, and labeling formats. It has individual labels and controls for kernel objects and services allowing fine-grained control over such abstractions including: file systems, directories, files, open file descriptions, sockets, messages, network interfaces, and use of capabilities. Additionally, SELinux has configurable default behavior that allows the security mechanisms to function transparently to applications.

The security model chosen for the prototype SELinux security server has proven to be a very effective model for security. Using its combination of IBAC, RBAC and TE, the SELinux security server provides flexible support for a wide range of security policies. With the SELinux security server, it is possible to configure the system to meet many security requirements. The flexibility of SELinux, allows the security server to be modified or replaced as needed without impacting the rest of the kernel.

The example security policy configuration released with SELinux serves as an example of how a number of important security objectives may be met using the prototype's security model. The flexibility of the security policy mechanism attained through the configuration files enables the policy to be easily modified and extended, allowing customization as might be required for any given installation. Hence, many security policies can be supported with the same base system.

Type Enforcement has proven a valuable security policy abstraction that has made the SELinux prototype a better system. Its advantages over the traditional approaches to MAC have led to a security policy that better protects the system. The potential that the TE access matrix could become quite complex is a possible downside to TE, but the benefits that TE offers should far outweigh this. Experience with SELinux shows that a realistic policy can be constructed that greatly improves security. Complexity can be managed through the distribution of base security policies with the system that allow individual installations to customize the policy as needed rather than start from scratch. Complexity can be further managed through policy specification language enhancements and the development of policy specification and analysis tools.

The SELinux prototype is very much a work in progress. SELinux was never intended to be a complete secure system. Instead, the purpose of SELinux was to serve as an example of how strong, flexible MAC could be added to a mainstream operating system to greatly improve the security of the system. SELinux succeeds in doing this for Linux. The Linux Security Module project [2] is an important effort to bring MAC to Linux. Success in its goal of creating the general security interface for the Linux kernel will enable Linux users to realize not only the security benefits of SELinux but also those from other security projects.

## Availability

The Security-Enhanced Linux software is available under the GNU General Public License (GPL) at `http://www.nsa.gov/selinux`.

## Acknowledgments

We thank Timothy Fraser for his contributions to the example policy configuration.

## References

[1] Linux Intrusion Detection System. http://www.lids.org.

[2] Linux security module. http://lsm.immunix.org.

[3] Medusa DS9. http://medusa.fornax.sk.

[4] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.

[5] D. E. Bell and L. J. La Padula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.

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

[7] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[8] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, Apr. 1987.

[9] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the USENIX 14th Systems Administration Conference (LISA)*, New Orleans, LA, Dec. 2000.

[10] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Oct. 1992.

[11] S. Hallyn and P. Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Oct. 2000.

[12] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. Technical report, NSA and NAI Labs, Oct. 2000.

[13] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[14] 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.

[15] A. Ott. Rule Set Based Access Control as proposed in the Generalized Framework for Access Control approach in Linux. Master's thesis, University of Hamburg, Nov. 1997. pp. 157. http://www.rsbac.org/papers.htm.

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

[17] 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.

[18] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, California, 1996.