A Security Policy Configuration for the Security-Enhanced Linux

Stephen Smalley, NAI Labs, sds@tislabs.com
Timothy Fraser, NAI Labs, tfraser@tislabs.com

February 2001

Contents

1 Introduction

The National Security Agency's Information Assurance Research Office is integrating a flexible mandatory access control architecture called *Flask* into the Linux operating system [1]. The Secure Execution Environments (SEE) group at NAI Labs is developing a Role-Based Access Control (RBAC) and Type Enforcement (TE) security policy configuration for this security-enhanced Linux system using the security policy configuration language described in [1, Sec 3.4]. This configuration draws from a preliminary configuration developed by Secure Computing Corporation and from the prior Domain and Type Enforcement (DTE) configuration developed by the SEE group [2]. The configuration also includes contributions by researchers from MITRE and contributions by researchers from the NSA. The configuration is still under development, and there are many areas where it still requires significant work.

This paper describes the current state of this security policy configuration. The paper begins with an overview of the security policy configuration. It then discusses the details of the configuration for Type Enforcement, Role-

Based Access Control, users, constraints, and security contexts. A separate configuration used to initially set file security contexts is then described. Finally, the paper describes configuration extensions to support the installation of the system.

2 Overview

This section provides an overview of the security policy configuration. It explains the basic concepts used in the configuration. It describes the goals for the configuration. It also provides a high-level explanation of how the policy configuration addresses these goals.

The security policy configuration defines a set of Type Enforcement domains and types. Each process has an associated domain, and each object has an associated type. The policy configuration specifies the allowable accesses by domains to types and the allowable interactions among domains. It specifies what types (when applied to programs) can be used to enter each domain and the allowable transitions between domains. It also specifies automatic transitions between domains when certain types are executed. These transitions ensure that system processes and certain programs are placed into their own separate domains automatically.

The configuration also defines a set of roles. Each process has an associated role. 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. These roles are set by the `login` program. A separate `newrole` program was added to support role changes within a login session.

The policy configuration specifies the set of domains that can be entered by each 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, transitions to other domains may automatically occur to support changes in privilege. Often, these other domains are derived from the user's initial login domain. For example, the *user_t* domain transitions to the *user_netscape_t* domain and the *sysadm_t* domain transitions to the *sysadm_netscape_t* domain when the `netscape` program is executed to restrict the browser to a subset of the user's permissions.

The first goal of the security policy configuration is to control various forms of raw access to data. The policy configuration defines distinct types for kernel memory devices, disk devices, and `/proc/kcore`. It defines separate domains for processes that require access to these types, such as *klogd_t* and *fsadm_t*.

The second goal is to protect the integrity of the kernel. The policy configuration defines distinct types for the boot files, module object files, module utilities, module configuration files and *sysctl* parameters, and it defines separate domains for processes that require write access to these files. It defines separate domains for the module utilities, and it restricts the use of the module capability to these domains. It only allows a small set of privileged domains to transition to the module utility domains.

The third goal is to protect the integrity of system software, system configuration information and system logs. The policy configuration defines distinct types for system libraries and binaries to control access to these files. It only allows administrators to modify system software. It defines separate types for system configuration files and system logs and defines separate domains for programs that require write access.

The fourth goal is to confine the potential damage that can be caused through the exploitation of a flaw in a process that requires privileges, whether a system process or privilege-enhancing (setuid or setgid) program. The policy configuration places these privileged system processes and programs into separate domains, with each domain limited to only those permissions it requires. Separate types for objects are defined in the policy configuration as needed to support least privilege for these domains.

The fifth goal is to protect privileged processes from executing malicious code. The policy configuration defines an executable type for the program executed by each privileged process and only allows transitions to the privileged domain by executing that type. When possible, it limits privileged process domains to executing the initial program for the domain, the system dynamic linker, and the system shared libraries. The administrator domain is allowed to execute programs created by administrators as well as system software, but not programs created by ordinary users or system processes.

The sixth goal is to protect the administrator role and domain from being entered without user authentication. The policy configuration only allows transitions to the administrator role and domain by the `login` program, which requires the user to authenticate before starting a shell with the administrator role and domain. It prevents transitions to the administrator role and domain by remote logins to prevent unauthenticated remote logins via `.rhosts` files. A `newrole` program was added to permit authorized users to enter the administrator role and domain during a remote login session, and this program re-authenticates the user.

The seventh goal is to prevent ordinary user processes from interfering with system processes or administrator processes. The policy configuration only allows certain system processes and administrators to access the *procfs* entries of processes in other domains. It controls the use of *ptrace* on other processes, and it controls signal delivery between domains. It defines separate types for the home directories of ordinary users and the home directories of administrators. It ensures that files created in shared directories such as `/tmp` are separately typed based on the creating domain. It defines separate types for terminals based on the owner's domain.

The eighth goal is to protect users and administrators from the exploitation of flaws in the `netscape` browser by malicious mobile code. The policy configuration places the browser into a separate domain and limits its permissions. It defines a type that users can use to restrict read access by the browser to local files, and it defines a type that users can use to grant write access to local files.

3 TE Configuration

In a traditional Type Enforcement (TE) policy, each subject is labeled with a domain, and each object is labeled with a type. The Flask security server merges the concepts of a domain and a type into a single type abstraction. A "domain" in Flask is simply a type that can be associated with a process. A type may be used both as a domain for a process and as a type for an object. For example, in the Linux implementation, the process-specific subdirectories in `/proc` are labeled with the security context of the corresponding process, so each domain is also used as the type of these pseudo files.

This section describes the Type Enforcement (TE) configuration contained in the `all.te` file. This file is automatically generated from a collection of files. The section begins by discussing the global macros defined for the TE configuration. It then describes a set of attributes used to group related types and domains together. The types and domains defined in the configuration are then individually discussed. Finally, the assertions that are checked after evaluating the TE configuration are described.

3.1 Global Macros

The `macros.te` file contains global macros used throughout the configuration for common groupings of classes and permissions and for common sets of rules. This subsection describes the macros defined in this file. These macros are used to ease specification of the configuration. The macros are expanded by the `m4` macro processor.

3.1.1 Class and Permission Macros Several macros are defined for groupings of file-related classes. The *dir_file_class_set* macro expands to the directory class and all of the file classes. The *file_class_set* macro expands to all file classes. The *notdevfile_class_set* macro expands to all file classes except for device special files, and the *devfile_class_set* macro expands to the device special file classes. These macros are used in access vector rules, type transition rules, and access vector assertions in the TE configuration. They are also used in the constraints configuration.

Several macros are defined for groupings of file permissions. The *stat_file_perms* macro expands to the permissions required to call *stat* or *access* on a file. This macro is useful in granting domains the ability to test for the existence of a file or *stat* files for a directory listing without granting any further accesses.

The *x_file_perms*, *r_file_perms*, *rx_file_perms* and *rw_file_perms* macros expand to the permissions required to execute a file, read a file, read and execute a file, and read and write a file, respectively. These macros are used to grant domains the ability to use existing files without granting them the ability to create, unlink, or rename them. Since it is desirable to strictly control execute access, file execute permission is only included in the

x_file_perms and *rx_file_perms* macros. A *rwx_file_perms* macro could be added, but most domains are not allowed to execute programs that they can write. It would be useful to add a *ra_file_perms* macro to indicate read and append access for append-only files.

The *link_file_perms* macro expands to permissions for linking, unlinking and renaming a file. This macro allows name space operations to be separately authorized from other operations. The *create_file_perms* macro expands to permissions for creating, reading, writing, linking, renaming and unlinking a file. This macro does not include file execute permission, since most domains are not allowed to execute programs that they can write. It also does not include permissions for relabeling, since it is desirable to strictly control relabeling operations.

The *r_dir_perms*, *rw_dir_perms*, and *create_dir_perms* macros provide similar expansions for directories. These macros differ in that they use directory-specific permissions such as *search*, *add_name*, *remove_name*, *reparent*, and *rmdir*. Directory search permission is included in the macros that permit reading, since search and read access are typically not separated in the policy configuration. It would be useful to add a *ra_dir_perms* macro to indicate read and *add_name* access for append-only directories. It might also be useful to add a *link_dir_perms* macro.

A single macro is currently defined for socket classes. The *socket_class_set* macro expands to the set of all socket classes. This macro is currently only used in the constraints configuration. It would be useful to add a *notrawsocket_class_set* macro that only expands to datagram and stream socket classes, since raw sockets should be limited to privileged domains.

The *rw_socket_perms* and *create_socket_perms* macros expand to permissions for reading and writing sockets and for creating, reading and writing sockets. These macros can be used for datagram or raw sockets. The *rw_stream_socket_perms* and *create_stream_socket_perms* macros are equivalent macros for stream sockets. It might be useful to add variants of these macros that are specific to clients and servers.

The *inherit_fd_perms* macro expands to permissions for inheriting and using an open file description. The most common use of this macro is to grant a domain the ability to inherit and use open file descriptions from the domain that transitioned to it. It is also sometimes necessary to grant these permissions for open file descriptions that are inherited through multiple domain transitions. For example, the *rlogind_t* domain inherits descriptions created by *inetd_t* indirectly through *tcpd_t*. The *receive_fd_perms* macro expands to permissions for receiving an open file description through local socket IPC and subsequently using it.

The *mount_fs_perms* macro expands to permissions for mounting and unmounting file systems. The *signal_perms* macro expands to permissions for sending any signal. The *packet_perms* macro expands to permissions for sending and receiving network packets. This macro can be used with either the node class or the network interface class.

3.1.2 Rule Macros The *domain_trans* macro expands to access vector rules that grant a parent domain the ability to transition to a child domain via a program type. In addition to defining the minimal set of access vector rules required to authorize the domain transition, this macro defines several rules that are not strictly required but are usually desired. For example, the macro grants the parent domain permissions to reap the child domain when it exits. It also grants the child domain permissions to inherit and use open file descriptions from the parent domain. It might be useful to add a minimal domain transition macro that only contains the rules required to authorize the transition. The *domain_auto_trans* macro adds a type transition rule to the *domain_trans* macro so that the domain transition occurs automatically when the program type is executed by the parent domain.

The *file_type_trans* and *file_type_auto_trans* macros provide similar functionality for transitioning to a new file type when a file is created. The first macro expands to access vector rules that grant a domain the ability to create a file type in a directory type. This macro also defines more than the minimal set of access vector rules. For example, it also grants the domain the ability to remove names from the directory type and to unlink the file type. The macro also defines access vector rules to allow creation of any file class except for device special files. It might be useful to add a minimal variant of this macro that only contains the rules required to authorize the file creation and that requires the desired file classes to be

explicitly specified. The *file_type_auto_trans* macro adds a type transition rule to this macro so that the file type transition occurs automatically when the domain creates a file in the directory type.

The *uses_shlib* macro expands to access vector rules that grant a domain the ability to execute the system dynamic loaders and to execute code from the system shared libraries. The *can_exec* macro grants a domain the ability to execute a program type without transitioning into a new domain. The *can_exec_any* macro grants a domain the ability to execute any system program.

The *can_network* macro expands to access vector rules that grant a domain the ability to perform unrestricted network communication via UDP or TCP sockets. This macro grants the domain permissions to the default message types for each network interface so that the domain can communicate with systems that do not provide message labeling. When message labeling is provided, separate access vector rules must be defined for the pair of domains that are communicating. The *can_tcp_connect* and *can_udp_send* macros expand to access vector rules that authorize specific pairs of domains to communicate. Since Flask does not yet provide message labeling across the network, these macros are only necessary for communication across the loopback interface.

For UNIX domain IPC, the *can_unix_connect* and *can_unix_send* macros expand to access vector rules that authorize specific pairs of domains to communicate. These macros do not authorize the transfer of open file descriptions between domains, so additional rules must be defined in the configuration if that is desired.

The *can_sysctl* macro expands to access vector rules that grant a domain the ability to modify any sysctl parameters. It might be useful to separate permissions for the `modprobe` path from the other sysctl parameters, since this path is especially security-critical. The *can_create_pty* macro expands to a set of rules that allow a user or administrator domain to create and access pseudo terminals with a corresponding derived type. The *can_create_other_pty* macro expands to a set of rules that allow a domain to create and access pseudo terminals on behalf of another domain, as in the case of `gnome-pty-helper`.

3.2 Type Attributes

Each type can have an optional set of attributes associated with it. A type attribute is used to identify a set of types with a similar property. When a type attribute is used in a rule, it is expanded to the set of types with that attribute. Hence, type attributes can be used to conveniently group types together and express shared properties for all types with the attribute. By prefixing a type attribute with the tilde character, a rule can also be applied to all types that do not have the specified attribute. The policy language does not yet support a set difference operator for type attributes.

The *domain* attribute is used to identify all types that can be used as domains. The TE configuration uses this attribute in rules to grant every domain a standard set of permissions. This attribute is also used in rules to allow certain privileged domains to send signals to all processes and to inspect the `procfs` entries of all processes. An access vector assertion uses this attribute to verify that only types with the domain attribute can be entered by processes.

The *privuser* attribute is used to identify all domains that can change their user identity. The *privrole* attribute is used to identify all domains that can change their role. The *privowner* attribute is used to identify all domains that can label objects with other user identities. These restrictions are specified in the constraints configuration.

The *privlog* attribute is used to identify all domains that can communicate with `syslogd` through its Unix domain socket. This attribute is used in rules that grant the necessary file permissions to the corresponding socket file. It is also used in rules that grant the necessary socket permissions for communicating with *syslogd*. The *privmem* attribute is used to identify all domains that can access kernel memory devices. This attribute is used in an assertion that only these domains have read or write access to the memory device type.

The *exec_type* attribute is used to identify all file types that are used as entry point executables for domains. This attribute is used in the *can_exec_any* macro to allow general execute access to these programs, although the ability to transition to the corresponding domains is more restricted. It is also used in an access vector assertion to verify that entry point executables can only be modified, deleted, or renamed by administrators.

Several attributes are defined to identify all types used for a particular kind of object. For example, *file_type* is used to identify all file types, *fs_type* attribute is used to identify all file system types, and *netif_type* is used to identify all network interface types. These attributes are used in access vector rules such as a rule to allow all file types to be created in a file system type and a rule to allow the `initrc` scripts to configure all network interfaces.

The *pidfile* attribute is used to identify all file types that are used as PID files in `/var/run`. The *tmpfile* attribute is used to identify all files types that are used as temporary files in one of the `tmp` directories. The *sysadmfile* attribute is used to identify file types that are fully accessible by the system administrator domain (*sysadm_t*).

3.3 General Types

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

This section describes each general type defined in the configuration. Domains and their associated types are discussed in the next section. This section begins by discussing types defined for new security objects introduced by Flask. It then describes types for controlling access to devices, types for controlling access to files, and types for controlling access to network objects.

3.3.1 Security Types The `security.te` file contains declarations for types defined for new security objects introduced by Flask. The security server type, *security_t*, is used to control the ability to use most of the new security server system calls. The policy configuration grants every domain permissions to obtain SIDs for contexts and to get the list of active SIDs. The permission to obtain a context for a SID is based on the type associated with the particular SID rather than using the generic *security_t* type. The policy configuration grants every domain this permission to every type, so the ability to obtain the security context associated with any SID is also unrestricted.

The policy configuration type, *policy_config_t*, is used to control access to the compiled policy configuration file (`/ss_policy`). The permission to load a new policy configuration on an operational system is also based on this type. This type can only be modified by the administrator. Stronger integrity protection could be provided by only allowing this type to be created or modified by the administrator through a specific program. Such a program could also require reauthentication to ensure that the policy configuration is not rewritten without user consent. Permission to load a new policy configuration is only granted between the administrator domain and this type.

The policy source type, *policy_src_t*, is used to control access to the policy configuration source files. This type can only be modified by the administrator. Since these source files have no standard location, the *file_contexts* configuration should be customized by each site to set the location of the policy configuration sources prior to relabeling the file system.

The file labels type, *file_labels_t*, is used to control access to the persistent label mapping stored in each file system. The mapping files are in the `...security` subdirectory at the root of each file system. This type can only be modified by the administrator. As with the policy configuration type, it might be desirable to provide stronger integrity protection for this type.

The inaccessible type, *no_access_t*, is a general type for files that are only accessible by administrators. This type is not currently used in the file context configuration.

3.3.2 Device Types The `device.te` file contains declarations for device types. The device directory type, *device_t*, is used to control access to the directory containing device special files. All domains are granted read and search permissions to directories of this type. This type is also used as the default type for files in this directory.

The null device type, *null_device_t*, is used to permit access to the null device. All domains are granted read and write permissions to this type. The random device type, *random_device_t*, is used to permit access to devices used to obtain random values. All domains are granted read permissions to this type.

The tty device type, *tty_device_t*, is used to control

access to tty devices. Tty devices are initially labeled with this type. The `login` program was modified to change the security context on the user terminal based on the user's security context. Derived types are defined for each user domain, *e.g. user_tty_device_t* and *sysadm_tty_device_t*, for this purpose. A distinct type, *devtty_t* is used for `/dev/tty` since it can be accessed by all domains.

The console device type, *console_device_t*, is used to control access to the console. Currently, all domains are granted read and write permissions to this type. This will be changed to only grant permissions for those domains that require access to the console device.

The memory device type, *memory_device_t*, is used to control raw access to memory. The klogd domain is allowed to read this type. The X server domain is currently allowed to read and write this type, although the portion of the X server that requires such access should be separated.

The fixed disk device type, *fixed_disk_device_t*, is used to control raw access to fixed disk devices. The removable device type, *removable_device_t*, is used to control raw access to removable devices. The file system administration program domain (used for programs such as `fsck` and `swapon`) is allowed to read and write these types. The administrator domain is currently allowed to directly read and write fixed disk devices to run `/sbin/lilo`, but this program will be moved into its own domain.

The clock device type, *clock_device_t*, is used to control access to the real time clock. The *initrc_t* domain is allowed to read and write this type. Note that a domain can set the system time without having access to this type.

The *misc_device_t* type is used to permit access to miscellaneous devices that have not yet been studied for proper control, e.g. `/dev/sequencer`, `/dev/dsp`, `/dev/audio`, `/dev/fb`. The user domains are allowed to read and write this type. These devices require further study to identify proper controls and may require changes to the `pam_console` module to set the security context on these device files based on the user security context.

The *psaux_t* type is used to control access to the `/dev/psaux` mouse device. The *initrc_t* domain is allowed to read this type for `kudzu`. The `gpm`, X server, and user domains are allowed to read and write this type. Properly controlling access to this device requires further study.

3.3.3 File Types The `file.te` file contains declarations for file types. At the end of the file, several rules are specified to define relationships among these file types.

The unlabeled type, *unlabeled_t*, is used to control access to files that do not yet support labeling. No domains are granted permissions to this type.

The default file system type, *fs_t*, is used to control access to the file system. This type is currently the only type defined for `ext2` file systems, and it is automatically applied to an unlabeled `ext2` file system when it is first mounted. All file types are allowed to be created in this file system type. All domains are allowed to get the attributes of this file system type. The *kernel_t*, *initrc_t*, and administrator domains are granted permissions to mount and unmount this type.

The default file type, *file_t*, is used to control access to files. This type is automatically applied to files in an unlabeled `ext2` file system when it is first mounted. All root directory types can be mounted on a directory with this type. The *initrc_t* and administrator domains are granted permissions to use directories with this type as mount points. Every domain is granted permissions to read directories and files of this type.

The root directory type, *root_t*, is used to control access to the root directory. All domains are allowed to read files and directories with this type. Only the administrator domains are granted permissions to modify this type.

The lost-and-found directory type, *lost_found_t*, is used to control access to the `lost+found` directories and files. Only the file system administration program domain and the administrator domains are granted permissions to this type.

The boot type, *boot_t*, is used to control access to the boot directory and its files. The administrator domains can modify this type. Since `/boot/kernel.h` is automatically generated during system initialization, a sep-

arate type, *boot_runtime_t*, is defined for this file. An automatic file type transition is defined for the *initrc_t* domain to create this type in the boot directory type. All domains are allowed to read these two types.

The tmp directory type, *tmp_t*, is used to control access to temporary directories. All domains are granted permissions to create and unlink files in these directories. To provide separation among temporary files, a separate derived type is defined for each domain that creates temporary files, and an automatic file type transition is defined for each domain to create the corresponding derived type in the tmp directory type.

The *etc_t* type is used to control access to system configuration information. This type can be read by any domain but can only be modified by the *passwd_t* and administrator domains. This type can also be executed by several domains. Since several configuration files are created during system initialization, an *etc_runtime_t* type is also defined. Automatic file type transitions are defined for the *init_t* and *initrc_t* domains to create files of this type in the *etc_t* directory type. Sendmail requires write access to the aliases database and the `/etc/mail` directory, so separate *etc_aliases_t* and *etc_mail_t* types are defined. The *sendmail_t* domain can read and write these two types, and can create new files in `/etc/mail`.

The *lib_t* type is used to control access to system libraries. All domains are allowed to read this type, but only administrator domains can modify it. Several domains can execute this type.

The *shlib_t* type is used to control access to system shared libraries.The *ld_so_t* type is used to control access to system dynamic loaders. All domains are allowed to read these two types, to execute programs with the *ld_so_t* type, and to execute code with the *shlib_t* type. Only administrator domains can modify these types. The set of domains will be reviewed to determine if they all require access to shared libraries.

The *bin_t* type is used to control access to system binaries. All domains are allowed to read this type, and several domains are allowed to execute it. Only administrator domains can modify it. The *sbin_t* type is used to control access to superuser system binaries. This type is identical to *bin_t* except that *init_t* can execute it for the `update` program.

The *man_t* type is used to control access to system manual page directories and files. All domains are allowed to read this type, and the administrator domains can modify it. The *system_crond_t* domain can also modify it to update the `whatis` files.

The *usr_t* type is used to control access to the `/usr` directory. The *src_t* type is used to control access to system sources. These types are currently equivalent to the root directory type. They are separately defined to allow distinct permissions to be granted in the future.

The *var_t* type is used to control access to the `/var` directory. This type is currently equivalent to the root directory type, but is separately defined to allow distinct permissions to be granted in the future. Separate types are defined for several subdirectories of `/var`: *catman_t*, *var_run_t*, *var_log_t*, *var_lock_t*, *var_lib_t*, *var_spool_t*, and *var_yp_t*. The *wtmp_t* type is defined for the `/var/log/wtmp` file. All domains can read these types.

All of these types can be modified by the administrator domains. The *catman_t* type can be read and modified by the user domains. The *var_run_t* type can be modified by daemons and by the *initrc_t* domain. The *var_log_t* type can be modified by *initrc_t*, *syslogd_t*, *crond_t*, *logrotate_t* and the login domains. The *var_lock_t* type can be modified by *initrc_t*, *system_crond_t*, and the local login domain. The *var_lib_t* type can be modified by *system_crond_t* and *logrotate_t*. The *var_yp_t* type can be modified by *ypbind_t*. The *wtmp_t* type can be modified by *init_t*, *initrc_t*, *getty_t*, *rlogind_t*, *utempter_t*, and the domains for `gnome-pty-helper` and `login`.

To provide separation among files in `/var/log`, derived types are defined for some of the domains that create files in this directory, and the `wtmp` file is assigned a separate type. The `logrotate` program was modified to preserve the security contexts on the log files in this directory.

To provide separation among files in `/var/run`, derived types are defined for each domain that creates files in this directory. Consequently, the pid files are individually labeled based on the corresponding domain, and the `utmp` file is labeled with the *initrc_var_run_t* derived type. The *initrc_t* domain is allowed to read and unlink the derived types for the pid files for shutting down the system. Domains for `init`, `getty`, `rlogind`,

`utempter`, `gnome-pty-helper`, `su` and `login` are granted read and write permissions to the `utmp` file.

The `/var/spool` directory is further refined into separate types for several of its subdirectories: *at_spool_t*, *cron_spool_t*, *lpd_spool_t*, *mail_spool_t*, and *mqueue_spool_t*. All of these types can be read or modified by the administrator domains. Each of the spool types can be accessed by the domains for the corresponding daemon and client programs. The login domains can test for the existence of mail spool files, and the user domains can read and write mail spool files. Derived types have been defined for several of these spool types to provide separation between spool files created by different user domains.

3.3.4 Procfs Types The `procfs.te` file contains declarations for types used for the pseudo files in `/proc`. The *proc_t* type is the type for the `/proc` directory and its files. All domains are allowed to read this type. Due to the highly sensitive nature of the `kmsg` and `kcore` files, separate types are defined for these files: *proc_kmsg_t* and *proc_kcore_t*. Only the domain for `klogd` is allowed to read the *proc_kmsg_t* type. Currently, no domain is allowed to read the *proc_kcore_t* type.

The process-specific subdirectories of `/proc` are labeled with the domain of the corresponding process. Each domain is allowed to read files labeled with the domain. The *initrc_t* and administrator domains are allowed to read files labeled with any domain.

The *sysctl_t* type is the type for the `/proc/sys` directory and its files. A separate type is defined for several of the subdirectories of `/proc/sys`: *sysctl_fs_t*, *sysctl_kernel_t*, *sysctl_net_t*, *sysctl_vm_t*, and *sysctl_dev_t*. Since the `modprobe` path is especially security-critical, a separate type, *sysctl_modprobe_t*, is defined for `/proc/sys/kernel/modprobe`. These types are also used to control the use of the *sysctl* system call. All domains are allowed to read these types. Only the *initrc_t* domain and the administrator domains are allowed to write these types.

3.3.5 Devpts Types The `devpts.te` file contains declarations for types used for the pseudo files related to `/dev/pts`.

The *ptmx_t* type is used to control access to the `/dev/ptmx` pty master multiplex device. The `rlogind` domain and user domains are allowed to read and write this type. The *devpts_t* type is the type for the `/dev/pts` directory. All domains are allowed to read this type. Pty files in `/dev/pts` are labeled with a type derived from the domain of the creating process. Each domain is granted access to its own ptys. Ptys created by `rlogind` are labeled with the *rlogind_devpts_t* type. The `login` program was modified to relabel the user terminal based on the user's security context. Consequently, ptys are relabeled by `login` to a derived type, *user_devpts_t* or *sysadm_devpts_t*.

3.3.6 NFS Types The `nfs.te` file contains declarations for types used for files from an NFS server. At the end of the file, several rules are specified to define relationships among these NFS file types.

The *nfs_t* type is the default type for NFS file systems and their files. A separate type can be defined for the files provided by each NFS server, as described in Section 7.3. The *nfs_clipper_t* type is an example type for NFS files mounted from a host named *clipper*. Currently, both of these types can be read and written by all domains.

3.3.7 Network Types The `network.te` file contains declarations for types used for network objects. At the end of the file, several rules are specified to define relationships among these network object types.

The *any_socket_t* type is the default destination socket type for UDP or raw IP traffic. The *can_network* macro grants the domain permission to send to this socket type. This macro is applied to any domain that uses the network.

The *icmp_socket_t* type is the type of the kernel socket used to send ICMP messages. This socket type is allowed to send and receive raw IP messages. The *tcp_socket_t* type is the type of the kernel socket used to send TCP resets. This socket is allowed to send and receive TCP messages. No domain is granted permissions to these socket types since they are only used internally by the kernel.

The *port_t* type is the default type for INET port numbers. All domains are allowed to bind port numbers with this type. Separate types are defined for several port numbers. Only the *lpd_t* domain is allowed to bind

printer_port_t. Only the *sendmail_t* domain is allowed to bind *smtp_port_t*. No domain is currently allowed to bind *http_port_t*. The *inetd_t* domain is allowed to bind the other types (*ftp_port_t*, *telnet_port_t*, *rlogin_port_t*, *rsh_port_t*). Hence, these types could be collapsed into a single *inetd_port_t* type. Port types are associated with specific port numbers through the network context configuration described in Section 7.3.

The *netif_t* type is the default type for network interfaces. The *netmsg_t* type is the default type for unlabeled messages received on network interfaces. Separate pairs of types are defined for several network interfaces: *netif_eth0_t* and *netmsg_eth0_t*, *netif_eth1_t* and *netmsg_eth1_t*, and *netif_lo_t* and *netmsg_lo_t*. Network interface types are associated with specific network interface names through the network context configuration described in Section 7.3. Permissions are granted for each unlabeled message type to be received on the corresponding network interface type. The *initrc_t* and administrator domains are allowed to configure any network interface. Several domains are allowed to get the configuration of any network interface. The *can_network* macro grants the domain permissions to send and receive on any network interface.

The *node_t* type is the default type for nodes. The *node_lo_t* type is the type for the loopback address. The *node_internal_t* type is the type for nodes on the local area network. Any of the unlabeled message types are allowed to be received from any node type. The *can_network* macro grants the domain permissions to send to any node type. Node types are associated with specific network addresses through the network context configuration described in Section 7.3.

3.4 Domains

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

This section describes each domain defined in the configuration. This section begins by discussing rules that are applied to every domain. It then describes the domains defined for system processes. Domains for user programs are then discussed. The section then describes domains for user login sessions.

3.4.1 Every Domain The `domains/every.te` file contains rules that apply to every domain. Each domain can send `SIGCHLD` to `init`. Each domain can access other processes in the same domain, e.g. each domain can send any signal to other processes in the same domain. Process-specific files in `/proc` can be accessed by any process with the same domain. Each domain is allowed to access open file descriptions, pipes, and sockets created by processes in the same domain.

Each domain is allowed to obtain SIDs for security contexts and to obtain the list of active SIDs. Each domain can obtain the security context for any SID.

Each domain can get the attributes for any file system type. Each domain has read access to the `procfs` types except for the *proc_kmsg_t* and *proc_kcore_t* types. Each domain has read access to most of the system file types, e.g. *file_t*, *root_t*, *usr_t*, *lib_t*, etc. Certain system file types are intentionally excluded from this general read access, such as lost-and-found directories (*lost_found_t*) and protected spool directories (e.g. *cron_spool_t*). Each domain can add and remove files from *tmp_t* directories.

Every domain is granted the ability to execute code from the system shared libraries and to execute the system dynamic loader. Since many domains only require execute access to these types and to their entry point executable, permission to execute other system binary types is not granted to all domains.

Each domain can read and write `/dev/tty`, `/dev/null`, and the random number devices. Currently, every domain is also allowed to read and write the console device, but this will be changed to only grant access to those domains that require such access.

Currently, every domain is allowed to create and use NFS files. Every domain is also currently allowed to use the network, bind to port numbers with the default port type, and communicate with portmap. These rules will be replaced with specific rules in the appropriate files granting these permissions to only those domains that require them.

3.4.2 System Domains The `domains/system` subdirectory contains a separate file for each domain used for a system process.

The *kernel_t* domain (`kernel.te`) is the domain of process `0` and the kernel threads started by it. No domain can transition to this domain. This domain is granted permissions for mounting and unmounting file systems and for searching the persistent label mapping. This domain automatically transitions to the *init_t* domain upon executing the `init` program.

The *kernel_t* domain is also the target type when checking permissions in the system class. This latter use of the *kernel_t* domain can be eliminated. The system permissions seem to be obsoleted by the capability permissions, so they can probably be completely eliminated. If the system permissions are retained, the calling process domain could be used instead as the target type, as with the capability permissions.

The *kmod_t* domain (`kmod.te`) is the domain of the kernel module loader. No domain can transition to this domain, so it can only be entered by the kernel. This domain can use the *sys_module* capability. It can execute `modprobe`, `insmod`, and shell commands from `conf.modules`. It can read `conf.modules`, `modules.dep`, and the module object files. It can signal any domain so that any process can wait on a kernel module loader thread.

The *init_t* domain (`init.te`) is the domain of the `init` process. Only the *kernel_t* domain can transition to this domain. The *init_exec_t* type is the type of the entry point executable for this domain. The *initctl_t* type is the type for `/dev/initctl`, a named pipe created by `init` for receiving communications. The *sulogin_exec_t* type is the type of the `sulogin` program used for authentication for single-user mode. The *init_t* domain can create `/dev/initctl` and `/etc/ioctl.save`. It can also modify `utmp` and `wtmp`. This domain can directly run the `update` program. All processes can be killed by this domain. It automatically transitions to *initrc_t* when it executes one of the `rc` scripts. It automatically transitions to *getty_t* when it executes `getty`. It automatically transitions to *sysadm_t* when it executes a shell or the `sulogin` program for single-user mode.

The *getty_t* domain (`getty.te`) is the domain of `getty`. Only the *init_t* domain is allowed to transition to this domain. The *getty_exec_t* type is the type of the entry point executable for this domain. The *getty_tmp_t* type is the type of temporary files created by this domain. This domain can update `utmp` and `wtmp`. It transitions to the *local_login_t* domain when it executes the `login` program.

The *initrc_t* domain (`initrc.te`) is the domain of the system `rc` scripts. Only the *init_t* domain can transition to this domain. The *initrc_exec_t* type is the type of the entry point executable for this domain. The *initrc_tmp_t* type is the type of temporary files created by this domain. The *initrc_var_run_t* type is the type of files created in `/var/run` by this domain.

The *initrc_t* domain can execute a variety of system programs, other `rc` scripts, and `telinit`. It can communicate with the *init_t* domain through `/dev/initctl`. It can examine all processes in `procfs` and send signals to any process. It can mount and unmount file systems of any type and configure any network interface. It can create various system runtime files. It can read and unlink PID files. This domain can set values in `/proc/sys`. It can use the network.

The *initrc_t* domain transitions to a corresponding daemon domain when it executes each system daemon. It transitions to the corresponding module utility domain when it executes a module utility. It transitions to the *fsadm_t* domain when it executes `fsck` and `swapon`. It transitions to the *ifconfig_t* domain when it executes `ifconfig`.

The *klogd_t* domain (`klogd.te`) is the domain of the kernel log daemon. Only the *initrc_t* domain can transition to this domain. The *klogd_exec_t* type is the type of the entry point executable for this domain. The *klogd_tmp_t* type is the type of temporary files created by this domain. The *klogd_var_run_t* type is the type of files created in `/var/run` by this domain. This domain can read `/proc/kmsg` and `/dev/mem`.

The *syslogd_t* domain (`syslogd.te`) is the domain of the system log daemon. Only the *initrc_t* domain can transition to this domain. The *syslogd_exec_t* type is the type of the entry point executable for this domain. The *syslogd_tmp_t* type is the type of temporary files created by this domain. The *syslogd_var_run_t* type is the type of files created in `/var/run` by this domain. The *devlog_t* type is used for `/dev/log`, a Unix domain socket

created by `syslogd` for receiving log messages. Domains with the *privlog* attribute can read and write this socket and can communicate with `syslogd`. The *syslogd_t* domain can modify log files. It can create and bind to `/dev/log`.

The *crond_t* domain (`crond.te`) is the domain of a daemon used to run scheduled commands. Only the *initrc_t* domain can transition to this domain. The *crond_exec_t* type is the type of the entry point executable for this domain. The *crond_tmp_t* type is the type of temporary files created by this domain. The *crond_var_run_t* type is the type of files created in `/var/run` by this domain. The *cron_log_t* type is the type of the cron log file. This domain can read from `/var/spool/cron` and it can read system and user crontab files. This domain transitions to *user_mail_t* when it executes `sendmail` for mailing output from cron jobs.

The `crond` program was changed to transition to a default security context for each user before executing any jobs for the user. The cron security contexts are specified in the `/etc/security/cron_context` file. The domains for these security contexts can be defined using the *crond_domain* macro from `crond.te`. This macro defines a derived domain for a user domain that can be used for cron jobs created by users in that domain. The use of a derived domain allows the policy to grant different permissions to user cron jobs than to an interactive user session.

Since crontab files are not directly executed, `crond` must ensure that the crontab file has a context that is appropriate for the context of the user cron job. The `crond` program was changed to perform an entrypoint permission check for this purpose. User crontab files are typed based on the domain that ran the `crontab` program. The domains defined by *crond_domain* are granted entrypoint permission to this type.

A *system_crond_t* domain is defined for system cron jobs to separate the permissions needed by system cron jobs from the permissions needed by the daemon itself. This domain is specified in the `/etc/security/cron_context` file for the *system_u* user. The *system_crond_script_t* type is used for system crontab files, and the *system_crond_t* domain is granted entrypoint permission to this type. This domain transitions to *rmmod_t* when it executes `rmmod` for `/etc/cron.d/kmod`. It transitions to *logrotate_t* when it executes `logrotate`.

The *atd_t* domain (`atd.te`) is the domain of another daemon that runs scheduled commands. Only the *initrc_t* domain can transition to this domain. The *atd_exec_t* type is the type of the entry point executable for this domain. The *atd_tmp_t* type is the type of temporary files created by this domain. The *atd_var_run_t* type is the type of files created in `/var/run` by this domain. Currently, this domain can read and write `/var/spool/at`. A separate type will be defined for `/var/spool/at/spool`, which is used for output from the jobs. This domain and program will be revised in a similar manner to *crond_t*.

The *sendmail_t* domain (`sendmail.te`) is the domain of the mail daemon. Only the *initrc_t* domain can transition to this domain. The *sendmail_exec_t* type is the type of the entry point executable for this domain. The *sendmail_tmp_t* type is the type of temporary files created by this domain. The *sendmail_var_run_t* type is the type of files created in `/var/run` by this domain. The *sendmail_var_log_t* type is the type of files created in `/var/log` by this domain. The *sendmail_t* domain can use the network and can bind to the SMTP port. It can write to the aliases database, `/etc/mail`, the mail spool directory, and the mail queue directory. The `sendmail` program is being analyzed to determine appropriate control points to insert transitions to derived domains for users so that its privileges are properly limited when acting on behalf of users.

The *lpd_t* domain (`lpd.te`) is the domain of the printer daemon. Only the *initrc_t* domain can transition to this domain. The *lpd_exec_t* type is the type of the entry point executable for this domain. The *lpd_tmp_t* type is the type of temporary files created by this domain. The *printer_t* type is used to control access to `/dev/printer`, a Unix domain socket created by `lpd`. This domain can use the network and bind to the network printer port. This domain can read and write `/var/spool/lpd`. Currently, this domain can directly execute filters in the spool directory or in system program directories. It may be desirable to transition to a separate domain when executing filters. For local printing, permissions will need to be added to local printer devices.

Since the `lpr` command can be used to create a symbolic link to the file rather than copying it into the spool directory, the *lpd_t* domain will either need to be granted permissions to read a variety of file types or it will need to transition to a default security context for the user prior to reading the file. The existing `lpd` program attempts to prevent abuse of its superuser privileges by checking that the device and inode number of the actual file are the same as when the link was created by `lpr`. However, this does not guarantee that the file is the same.

The *gpm_t* domain (`gpm.te`) is the domain of the console mouse server. Only the *initrc_t* domain can transition to this domain. The *gpm_exec_t* type is the type of the entry point executable for this domain. The *gpm_tmp_t* type is the type of temporary files created by this domain. The *gpm_var_run_t* type is the type of files created in `/var/run` by this domain. The *gpmctl_t* type is used for `/dev/gpmctl`, a Unix domain socket created by `gpm` for communications. This domain can create and bind to `/dev/gpmctl`. It can access `/dev/psaux`. Permissions are not yet defined to allow client domains to communicate with this domain.

The *xfs_t* domain (`xfs.te`) is the domain of the X font server. Only the *initrc_t* domain can transition to this domain. The *xfs_exec_t* type is the type of the entry point executable for this domain. The *xfs_tmp_t* type is the type of temporary files created by this domain. This domain can create and bind to sockets in `/tmp/.font-unix`. The X server program domains can communicate with this domain.

The *apmd_t* domain (`apmd.te`) is the domain of the `apmd` daemon. Only the *initrc_t* domain can transition to this domain. The *apmd_exec_t* type is the type of the entry point executable for this domain. The *apmd_var_run_t* type is the type of files created in `/var/run` by this domain. The *apm_bios_t* type is the type of `/dev/apm_bios`. This domain can access `/dev/apm_bios`.

The *cardmgr_t* domain (`cardmgr.te`) is the domain of the `cardmgr` daemon. Only the *initrc_t* domain can transition to this domain. The *cardmgr_exec_t* type is the type of the entry point executable for this domain. The *cardmgr_var_run_t* type is the type of files created in `/var/run` by this domain. The *cardmgr_dev_t* type is the type of character devices created by this domain in `/tmp`. The *cardmgr_lnk_t* type is the type of symbolic links created by this domain in `/dev`. This domain can execute a shell and system programs. It can transition to the *insmod_t* domain and the *rmmod_t* domain by executing the corresponding module utility. It can transition to the *ifconfig_t* domain by executing the `ifconfig` program. This domain requires further review.

The *inetd_t* domain (`inetd.te`) is the domain of the Internet superserver. Only the *initrc_t* domain can transition to this domain. The *inetd_exec_t* type is the type of the entry point executable for this domain. The *inetd_tmp_t* type is the type of temporary files created by this domain. The *inetd_var_run_t* type is the type of files created in `/var/run` by this domain. This domain can use the network and can bind to a variety of port numbers. It transitions to the *tcpd_t* domain when it executes `tcpd`. It transitions to the *inetd_child_t* domain when it executes other daemons.

The *inetd_child_t* domain (`inetd.te`) is a general domain for daemons started by *inetd* or *tcpd* that do not have their own individual domains yet. Either *inetd_t* or *tcpd_t* can transition to this domain. The *inetd_child_exec_t* type is the type of the entry point executable for this domain. The *inetd_child_tmp_t* type is the type of temporary files created by this domain. The *inetd_child_var_run_t* type is the type of files created in `/var/run` by this domain. This domain is only a stub.

The *tcpd_t* domain (`tcpd.te`) is the domain of the TCP wrapper daemon. Only the *inetd_t* domain can transition to this domain. The *tcpd_exec_t* type is the type of the entry point executable for this domain. The *tcpd_tmp_t* type is the type of temporary files created by this domain. This domain can use the network and can use TCP sockets inherited from *inetd_t*. It transitions to the *rlogind_t* domain when it executes `rlogind` or `telnetd`. It transitions to the *rshd_t* domain when it executes `rshd`. It transitions to the *ftpd_t* domain when it executes `ftpd`. It transitions to the *inetd_child_t* domain when it executes other daemons.

The *rlogind_t* domain (`rlogind.te`) is the domain of the daemons for telnet and remote login. Only the *tcpd_t* domain can transition to this domain. The *rlogind_exec_t* type is the type of the entry point executable for this domain. The *rlogind_tmp_t* type is the type of temporary files created by this domain. This do-

main can use the network and can use TCP sockets inherited from *inetd_t*. It can create ptys. It can modify `utmp` and `wtmp`. It transitions to the *remote_login_t* domain when it executes `login`.

The *rshd_t* domain (`rshd.te`) is the domain of the `rshd` daemon. Only the *tcpd_t* domain can transition to this domain. The *rshd_exec_t* type is the type of the entry point executable for this domain. This domain can use the network and can use TCP sockets inherited from *inetd_t*. The `rshd` program was modified to read an initial security context for the user from a `/etc/security/rsh_contexts` configuration file and to run the shell with this security context. It can only transition to the *user_t* domain, so it can not be used to enter an administrator domain. This restriction is to prevent entry to an administrator domain without authentication.

The *ftpd_t* domain (`ftpd.te`) is the domain of the `ftpd` daemon. Only the *tcpd_t* domain can transition to this domain. The *ftpd_exec_t* type is the type of the entry point executable for this domain. The *ftpd_var_run_t* type is the type of files created in `/var/run` by this domain. This domain can use the network and can use TCP sockets inherited from *inetd_t*. The `ftpd` program is being modified to transition to a configurable security context for the user after the user has been authenticated. The *ftpd_domain* macro is used to define derived domains for user ftp sessions.

The *ypbind_t* domain (`ypbind.te`) is the domain of the NIS binding daemon. The *portmap_t* domain is the domain of a daemon that maps RPC program numbers to port numbers. The *rpcd_t* domain is a general domain for other RPC daemons. Only the *initrc_t* domain can transition to these domains. These daemons have not yet been studied for proper permissions.

The *local_login_t* domain (`login.te`) is a domain for local logins. Only the *getty_t* domain can transition to this domain. The *login_exec_t* type is the type of the entry point executable for this domain. The *local_login_tmp_t* type is the type of temporary files created by this domain. This domain can use the network to perform NIS lookups. It can read and write `utmp`, `wtmp`, and `lastlog`. It can search the mail spool directory so that it can check for mail for the user. It can transition to any of the domains for user login sessions when it executes a shell. By default, it automatically transitions to the *user_t* domain when it executes a shell.

The `login` program was modified to provide a default login context for each user and to allow the user to specify a different context for the login session. The `login` program was also changed to relabel the user terminal with a security context derived from the user's security context. The `pam_console` module still needs to be modified to relabel other devices accordingly.

The *remote_login_t* domain (`login.te`) is a domain for remote logins. Only the *rlogind_t* domain can transition to this domain. This domain has a few differences from *local_login_t*. The *remote_login_tmp_t* type is the type of temporary files created by this domain. This domain can use ptys created by `rlogind`. It can only transition to the *user_t* domain, so it can not be used to enter an administrator domain. This restriction is to prevent unauthenticated remote logins by administrators via `.rhosts` files. A separate `newrole` program was added to support changing from *user_t* to *sysadm_t* after authenticating to permit remote users to enter the administrator domain after login.

3.4.3 User Program Domains The `domains/program` subdirectory contains a separate file for each domain used for a user program.

Types and domains for the privileged module utilities are defined in the `modutil.te` file. The *modules_conf_t* type is for the `/etc/conf.modules` configuration file. The *modules_dep_t* type is used for the `modules.dep` files. The *modules_object_t* type is used for the module object directories and files.

The *modprobe_t*, *depmod_t*, *insmod_t*, and *rmmod_t* domains are defined for the corresponding utilities, and each domain has a corresponding entry point executable type. The *initrc_t* and administrator domains can transition to these domains. Both the *cardmgr_t* domain and the *modprobe_t* domain can transition to the *insmod_t* or *rmmod_t* domains. The *crond_t* domain can transition to the *rmmod_t* domain for the `/etc/cron.d/kmod` crontab file.

The *modprobe_t* domain can execute shell commands from `conf.modules`. The *depmod_t* domain can create `modules.dep`. The *insmod_t* and *rmmod_t* domains can use the *sys_module* capability.

When executed by the kernel module loader, the `modprobe` and `insmod` programs remain in the *kmod_t* domain. This allows the security policy to distinguish between permissions granted to the kernel module loader and permissions granted to module utilities executed by user processes. For example, the security policy could be configured to prohibit any transitions to the *modprobe_t* and *insmod_t* domains while still allowing the kernel module loader to function.

The *logrotate_t* domain (`logrotate.te`) is the domain for the `logrotate` program. Only the *system_crond_t* domain and the administrator domains can transition to this domain. The *logrotate_exec_t* type is the type of the entry point executable for this domain. The *logrotate_tmp_t* type is the type of temporary files created by this domain. This domain can create, rename and truncate log files, and it can set the appropriate security context and Unix ownership. It can read the PID files, search `/proc`, and signal any domain in order to notify daemons of changes in log files. It can update *var_lib_t* for `/var/lib/logrotate.status`. The `logrotate` program was modified to preserve the security context of log files.

The *fsadm_t* domain (`fsadm.te`) is the domain for disk and file system administration programs such as `fsck` and `swapon`. Only the *initrc_t* domain and the administrator domains can transition to this domain. The *fsadm_exec_t* type is the type of the entry point executable for this domain. The *fsadm_tmp_t* type is the type of temporary files created by this domain. This domain can write to `/etc/mtab` and it can access the raw disk devices.

The *ifconfig_t* domain (`ifconfig.te`) is the domain for the `ifconfig` program. Only the *initrc_t* domain, *cardmgr_t* domain, and the administrator domains can transition to this domain. The *ifconfig_exec_t* type is the type of the entry point executable for this domain. This domain can use the *sys_module* capability to load network interface modules and it can configure the network interfaces.

The *utempter_t* domain (`utempter.te`) is the domain for the `utempter` program. Any of the user login domains can transition to this domain. The *utempter_exec_t* type is the type of the entry point executable for this domain. The *utempter_t* domain can read and write `utmp` and `wtmp`, allowing the `utempter` program to log the beginnings and ends of user sessions on behalf of the `xterm` virtual terminal program.

The *passwd_t* domain (`passwd.te`) is the domain for changing passwords and other user information. Any of the user login domains can transition to this domain. The *passwd_exec_t* type is the type of the entry point executable for this domain. This domain can read and write `/etc` and `/etc/auth`. It can also test for the existence of a shell and read `utmp`.

Since the ordinary programs for changing passwords and other user information (`passwd`, `chfn`, `chsh`) allow the superuser to change any user's information, it was necessary to interpose a wrapper program to prevent this behavior, as in [2]. The wrapper programs (`spasswd`, `schfn`, `schsh`) only call the real programs if the Flask user identity of the calling process is the same as the Unix real user identity, and these programs do not pass any arguments to the real programs. These wrapper programs will be changed to pass unprivileged arguments. Since the *passwd_t* domain can only be entered through the wrapper programs, an unprivileged user login domain cannot bypass the wrapper programs. Administrator domains can directly execute the regular programs and change other users' information as the superuser.

The X server program domains are *user_xserver_t* and *sysadm_xserver_t*. These domains are defined using the *xserver_domain* macro in `xserver.te`. The *xserver_exec_t* type is the type of the entry point executable for these domains. The *user_xserver_tmp_t* and *sysadm_xserver_tmp_t* types are the types of temporary files created by these domains. Each X server domain can create and bind to a socket in `/tmp` with the corresponding temporary type. It can connect to the X font server domain. It can receive connections from the corresponding user login domain. Currently, it can read and write memory devices, although the portion of the X server that requires this access should be separated. It can execute a variety of system programs.

The `lpr` domains are *user_lpr_t* and *sysadm_lpr_t*. These domains are defined using the *lpr_domain* macro in `lpr.te`. These domains are used for the client printing commands `lpr`, `lpq`, and `lprm`. The *lpr_exec_t* type is the type of the entry point executable for these

domains. The *user_lpr_tmp_t* and *sysadm_lpr_tmp_t* types are the types of temporary files created by these domains. Each domain can create spool files with a derived type in `/var/spool/lpd`. It can connect to `lpd` and send SIGHUP to the daemon. It can read from pipes created by the user login domain.

The `sendmail` program domains are *user_mail_t* and *sysadm_mail_t*. These domains are defined using the *mail_domain* macro in `mail.te`. The *sendmail_exec_t* type is the type of the entry point executable for these domains. The *user_mail_tmp_t* and *sysadm_mail_tmp_t* types are the types of temporary files created by these domains. These domains share many of the same permissions as the *sendmail_t* system domain. They can also read temporary files created by the user login domain for sending mail and they can write to the user domain's home directory type to create the `dead.letter` file.

Currently, the `mail` program does not run in a separate domain from the user login domains, since it does not require any special permissions to access the mail spool files. To prevent the superuser from reading and writing all mail spool files, the individual spool files could be created with a type based on the default login domain for the user. Alternatively, a wrapper for the `mail` program could be created with its own domain to ensure that the program is only used to access the mail spool file for the Flask user identity.

The `gnome-pty-helper` program domains are *user_gph_t* and *sysadm_gph_t*. These domains are defined using the *gph_domain* macro in `gnome-pty-helper.te`. The *gph_exec_t* type is the type of the entry point executable for this domain. The `gnome-pty-helper` program creates new pseudo-terminals for instances of the `gnome-terminal` virtual terminal program running in the user login domains, and logs the beginnings and ends of `gnome-terminal` sessions to `utmp` and `wtmp`. Each of the `gnome-pty-helper` domains supports this behavior by providing read and write access to the `/dev/ptmx` device, `utmp`, and `wtmp`, and by permitting the passing of open file descriptors to programs in the corresponding user login domains via local socket IPC.

The `su` domains are *user_su_t* and *sysadm_su_t*. These domains are defined using the *su_domain* macro in `su.te`. The *su_exec_t* type is the type of the entry point executable for these domains. Each `su` domain automatically reverts to the domain of the caller when it executes a shell. It can read the shadow password file for user authentication. It can update the `utmp` file. It can modify the user's `.Xauthority` file. Since the `su` program is most frequently used simply to obtain Unix privileges for administrative tasks by becoming the superuser, it seems to be undesirable to also change the Flask user identity, so only the Unix identity is changed.

The `netscape` domains are *user_netscape_t* and *sysadm_netscape_t*. These domains are defined using the *netscape_domain* macro in `netscape.te`. The *netscape_exec_t* type is the type of the entry point executable for these domains. These domains are limited to writing to a derived type: *user_netscape_rw_t* and *sysadm_netscape_rw_t*. The file contexts configuration uses the *user_netscape_rw_t* type for the `.netscape` directories, the `.mime.types` file and the `.mailcap` file. Users can also apply this type to other files or directories that should be writeable by `netscape`. These `netscape` domains are not allowed to read a different derived type: *user_netscape_noread_t* and *sysadm_netscape_noread_t*. Users can apply this type to files that should not be readable by `netscape`.

The `crontab` domains are *user_crontab_t* and *sysadm_crontab_t*. These domains are defined using the *crontab_domain* macro in `crontab.te`. The *crontab_exec_t* type is the type of the entry point executable for these domains. The *user_cron_spool_t* and *sysadm_cron_spool_t* types are the types for the crontab files created by these domains in `/var/spool/cron`.

3.4.4 User Login Domains The `domains/user` subdirectory contains a separate file for each domain used for an ordinary user login. The `domains/admin` subdirectory contains a separate file for each domain used for an administrator login. Currently, there is a single domain for ordinary users and a single domain for administrators.

The *user_t* domain is the initial login domain for unprivileged users. The *local_login_t*, *remote_login_t*, and *rshd_t* domains can transition to this domain. This domain is defined using the *user_domain* macro in `user.te`. The *shell_exec_t* type is the type of the entry

point executable for this domain. The *user_home_t* type is the type for home directories of ordinary users. The *user_tmp_t* type is the type of temporary files created by this domain. The *user_tty_device_t* type is the type of tty devices owned by this domain. The *user_devpts_t* type is the type of pty devices owned by this domain. This domain can use the network. It can execute a variety of system programs. It can read, write or execute files in its home directory type. It can transition to several of the user program domains when it executes the corresponding program.

The *sysadm_t* domain is the initial login domain for system administrators. The *init_t* and *local_login_t* domains can transition to this domain. This domain is defined using the *admin_domain* macro in `sysadm.te`. The *shell_exec_t* type is the type of the entry point executable for this domain. The *sysadm_home_t* type is the type for home directories of administrators. The *sysadm_tmp_t* type is the type of temporary files created by this domain. The *sysadm_tty_device_t* type is the type of tty devices owned by this domain. The *sysadm_devpts_t* type is the type of pty devices owned by this domain. This domain is allowed to perform administrative tasks such as running module utilities, mounting and unmounting file systems, configuring network interfaces, and running `telinit`. It can read and write all file types with the *sysadmfile* attribute. It can examine `procfs` for all processes and send signals to all processes. It can load new policy configurations and it can relabel files.

The file contexts configuration uses *user_home_t* as the type for `/home` and *sysadm_home_t* as the type for `/root`. This configuration must be customized to properly type the home directories for administrators and ordinary users of the site. Currently, all domains are granted read access to these types. Many domains require read access in order to read user dotfiles. The `mail` program domains are granted permission to write the corresponding user home directory type to create the `dead.letter` file. The `su` program domains are granted permission to update the `.Xauthority` file. Each user domain is granted permissions to read, write, and execute its own home directory type. The administrator domain is also granted permissions to read and write the ordinary user home directory type, but not to execute it.

Although a user may be authorized as an administrator, the user should still login in the *user_t* domain unless he is performing administrative tasks. Otherwise, the user may unintentionally abuse his privileges. Currently, the ability of an administrator to login in the *user_t* domain is complicated by the fact that the administrator's home directory has a separate type that is not writeable by the *user_t* domain. This problem will be solved either by adding support for multiple home directories for a user or by adding support for polyinstantiated directories.

3.5 Assertions

The `assert.te` file contains assertions that are checked after evaluating the entire TE configuration. These assertions can be used to detect errors in the configuration.

A few sample assertions are provided, but a thorough set of assertions has not yet been developed. Some of the sample assertions are that only certain domains can use the *sys_module* capability and that system software can only be modified by administrators.

An *assert_execute* macro is defined for generating assertions to verify that certain domains can only execute code from their entry point executable type, the system dynamic loader type, and the system shared library type. This macro is applied to a set of domains that should not require execute access to any other code.

4 RBAC Configuration

This section describes the Role-Based Access Control (RBAC) configuration contained in the `rbac` file. It begins by discussing each `m4` macro. It then describes each role.

4.1 Macros

Currently, there is only one macro defined for the RBAC configuration. The *role_auto_trans* macro expands to a role allow rule that authorizes a role transition and a role transition rule that causes the transition to occur automatically when a particular program type is executed. This macro is the RBAC equivalent to the TE *domain_auto_trans* macro.

4.2 Roles

The *object_r* role is a predefined role that is used for objects, since the role field in an object security context is not used in access decisions. Any type can be associated with this role. A role allow rule to this role should never be defined, since a process with this role could potentially enter any domain.

The *system_r* role is the role of system processes. Any of the TE system domains described in Section 3.4.2 can be associated with this role. The *sysadm_t* domain is authorized for this role so that `init` can enter this domain for single-user mode. The *fsadm_t*, *ifconfig_t*, and module program domains are also authorized for this role so that *initrc_t* can execute the corresponding programs.

The *user_r* role is the role of unprivileged user processes. The initial login domain for this role is the *user_t* domain. This role is also authorized for a variety of user program domains.

The *sysadm_r* role is the role of the system administrator. The initial login domain for this role is the *sysadm_t* domain. This role is also authorized for a variety of user program domains, including domains for `ifconfig`, `fsck`, and the module utilities.

These user roles can be entered at login. To support role changes during a login session, a `newrole` program was created. This program reauthenticates the user to ensure that the role change does not occur without consent by the user. The program transitions to the new role and to the initial login domain associated with that role. This program is run in the *newrole_t* domain that is authorized for role changes.

5 User Configuration

This section describes the user configuration contained in the `users` file. This configuration defines each user recognized by the security policy. It specifies the roles that can be associated with each user.

The *system_u* user is the user identity for system processes and objects. There should be no corresponding Unix identity for the Flask *system_u* user, and a user process should never be assigned the *system_u* user identity. The *system_r* role can be associated with this user identity.

The remaining users listed in this configuration correspond to Unix identities in the `/etc/passwd` file. These user identities are assigned to user processes when `login` creates the user shell. The *user_r* role can be associated with any user. The *sysadm_r* role can be associated with any system administrator.

Although a user may be authorized for an administrator role, the user should still login in the *user_r* role unless he is performing administrative tasks. Otherwise, the user may unintentionally abuse his privileges. Currently, the ability of an administrator to login in the *user_r* role is complicated by the fact that the administrator's home directory has a separate type that is not writeable by the *user_t* domain. This problem will be solved either by adding support for multiple home directories for a user or by adding support for polyinstantiated directories.

6 Constraints Configuration

This section describes the constraints configuration contained in the `constraints` file. This configuration defines additional restrictions on certain permissions. These restrictions are expressed as boolean expressions based on the relevant user identities, roles, and types.

Two constraints are defined for the process transition permission. The first constraint restricts the ability to transition to a different user identity to domains with the *privuser* type attribute. Only the `crond` and login domains need this attribute. The second constraint restricts the ability to transition to a different role to domains with the *privrole* type attribute. Only the `crond`, login domains, and the domain for the `newrole` program need this attribute.

Two constraints are defined for creating and relabeling objects. The first constraint restricts the ability to create or relabel files with a different owner to domains with the *privowner* attribute. The second constraint restricts the ability to create or relabel sockets with a different owner to domains with the *privowner* attribute. The administrator domain and the *logrotate_t* domain have this attribute.

7 Security Context Configuration

This section describes the security context configuration. It begins by discussing the security contexts for

initial SIDs. The contexts for unlabeled file systems are then described. This section concludes with a description of the contexts for network objects.

7.1 Initial SID Contexts

The initial SID context configuration is contained in the `initial_sid_contexts` file. This configuration specifies the security context for each SID that is predefined for system initialization.

A separate domain or type is defined for each initial SID so that the TE configuration can distinguish among the initial SIDs. The domains associated with the kernel, `init`, and kernel module loader SIDs are described in Section 3.4. The types associated with the other initial SIDs are described in Section 3.3.

All of the initial SID contexts use the *system_u* user identity, since they represent system processes and objects. The kernel SID, `init` SID, and kernel module loader SID use the *system_r* role since they are used for system processes. The initial SIDs for sockets (*any_socket*, *icmp_socket*, and *tcp_socket*) use the *system_r* role because sockets are treated as proxies for processes in the network access control model. The other initial SIDs use the *object_r* role since they represent objects.

7.2 File System Contexts

The unlabeled file system context configuration is contained in the `fs_contexts` file. This configuration specifies the security contexts to apply to an unlabeled file system when it is first mounted. If no entry is specified for the device, then the security contexts associated with the *fs* and *file* initial SIDs are used.

Currently, this configuration is unused. A single entry is specified as an example. The types are the same as for the *fs* and *file* initial SIDs. The *system_u* user identity and *object_r* role are used since these contexts represent system objects.

7.3 Network Contexts

The network object context configuration is contained in the `net_contexts` file. This configuration specifies the security contexts for port numbers, network interfaces, nodes, and NFS files. The types associated with these contexts are discussed in Section 3.3.7. These security contexts use the *system_u* user identity and the *object_r* role since they represent system objects.

By default, port numbers are labeled with the security context associated with the *port* initial SID. Separate security contexts are specified for port numbers that should be restricted to particular domains. Currently, security contexts are only defined for a few ports as examples. As discussed in Section 3.3.7, several of the types used in these security contexts can be reduced to a single *inetd_port_t* type.

The security contexts associated with the *netif* and *netmsg* initial SIDs are used by default for network interfaces. Separate security contexts can be specified for individual network interfaces to distinguish access to different interfaces. Currently, separate contexts are defined for the loopback interface, the `eth0` interface, and the `eth1` interface. However, these distinctions are not currently used by the TE configuration.

By default, the security context associated with the *node* initial SID is used for nodes. Separate security contexts can be specified for an address and mask pair to distinguish access to different nodes. Currently, separate contexts are defined for the localhost address and for all hosts with a particular prefix. However, these distinctions are not currently used by the TE configuration.

NFS filesystems and files are labeled with the security context associated with the *nfs* initial SID by default. Separate security contexts can be specified for an address and mask pair to distinguish access to different NFS servers.

8 File Contexts

This section describes the separate configuration used to set file security contexts. This configuration is contained in the `file_contexts` file. It specifies file security contexts based on pathname regular expressions. The `setfiles` program reads this configuration and labels files accordingly.

Since the file system layout varies considerably among different Linux distributions and even among different versions of a single Linux distribution, this configuration should be reviewed and customized before the initial relabeling of the file system. For example, the locations

of `syslogd` and `klogd` differ between RedHat 6.0 and RedHat 6.1. As mentioned in Section 3.3.1, the location of the policy sources (*policy_src_t*) should also be customized before the initial relabeling. Similarly, as mentioned in Section 3.3.3, the location for administrator and ordinary user home directories should be customized before the initial relabeling.

The types used in the configuration are described in Section 3.3. The *system_u* user identity and *object_r* role are used for all of the security contexts in this configuration, since they all represent system objects. If desired, a separate entry could be specified for each user home directory so that it is labeled with the user's identity. However, this is not necessary, since the user identity on the file is only used to determine the ability to relabel the file. Any files created subsequently by individual users will be created with the corresponding user identity.

9 Extensions for Installing

When the new kernel is first booted on a vanilla Linux system, all files are initially labeled with the security context associated with the *file* initial SID. Since the entry point executables are not labeled yet, appropriate domain transitions do not occur. Since all files are labeled with a single type, the permissions defined in the standard TE configuration are inadequate.

Consequently, a set of extensions to the standard policy configuration are defined for the initial boot and relabeling of file systems. The extended policy configuration is referred to as the *initial policy*. After booting with the initial policy, file systems are relabeled in accordance with the file contexts configuration and the standard policy is installed. The system is then rebooted for operational use.

The extensions to the policy configuration are contained in the `init.te` file. This file defines a new *initial_boot_t* domain. The *init_t* domain transitions to this domain when it executes any file. All system processes run in this domain during the initial boot. This domain is granted extensive permissions so that all system processes can perform their tasks before or after the relabeling. This domain can transition to the *sysadm_t* domain so that a user can login as the administrator. The user can then relabel file systems, install the standard policy, and reboot.

In addition to defining the new domain, this file extends the *kernel_t*, *init_t*, *kmod_t*, and *sysadm_t* domains so that they can function properly during the initial boot and relabeling. It also extends the *initrc_t* domain so that it can handle the reboot.

References

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

[2] 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.