Integrating Flexible Support for Security Policies into the Linux Operating System

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

Abstract

The protection mechanisms of current mainstream operating systems are inadequate to support confidentiality and integrity requirements for end systems. Mandatory access control (MAC) is needed to address such requirements, but the limitations of traditional MAC have inhibited its adoption into mainstream operating systems. The National Security Agency (NSA) worked with Secure Computing Corporation (SCC) to develop a flexible MAC architecture called Flask to overcome the limitations of traditional MAC. The NSA has implemented this architecture in the Linux operating system, producing a Security-Enhanced Linux (SELinux) prototype, to make the technology available to a wider community and to enable further research into secure operating systems. NAI Labs has developed an example security policy configuration to demonstrate the benefits of the architecture and to provide a foundation for others to use. This paper describes the security architecture, security mechanisms, application programming interface, security policy configuration, and performance of SELinux.

1 Introduction

End systems must be able to enforce the separation of information based on confidentiality and integrity requirements to provide system security. Operating system security mechanisms are the foundation for ensuring such separation. Unfortunately, existing mainstream operating systems lack the critical security feature required for enforcing separation: mandatory access control (MAC) [17]. Instead, they rely on discretionary access control (DAC) mechanisms. As a consequence, application security mechanisms are vulnerable to tampering and bypass, and malicious or flawed applications can easily cause failures in system security.

DAC mechanisms are fundamentally inadequate for strong system security. DAC access decisions are only based on user identity and ownership, ignoring other security-relevant information such as the role of the user, the function and trustworthiness of the program, and the sensitivity and integrity of the data. Each user has complete discretion over his objects, making it impossible to enforce a system-wide security policy. Furthermore, every program run by a user inherits all of the permissions granted to the user and is free to change access to the user's objects, so no protection is provided against malicious software. Typically, only two major categories of users are supported by DAC mechanisms, 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, so that a flaw in any one of these programs can be exploited to obtain complete system access.

By adding MAC mechanisms to the operating system, these vulnerabilities can be addressed. MAC access decisions are based on labels that can contain a variety of security-relevant information. A MAC policy is defined by a system security policy administrator and enforced over all subjects (processes) and objects (e.g. files, sockets, network interfaces) in the system. MAC can support a wide variety of categories of users on a system, and it can confine the damage that can be caused by flawed or malicious software.

Traditional MAC mechanisms have typically been tightly coupled to a multi-level security (MLS) [7] 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 [8, 9, 10]. 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. This limits the ability of the system 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.

To address the limitations of traditional MAC, the National Security Agency (NSA), with the help of Secure Computing Corporation (SCC), began researching new ways to provide strong mandatory access controls that could be acceptable for mainstream operating systems. An important design goal for the NSA 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. Through the development of two Mach-based prototypes, DTMach [12] and DTOS [20], the NSA and SCC developed a strong, flexible security architecture. Although high assurance was not a goal of the research, formal methods were applied to the design to help validate the security properties of the architecture [23, 24]. Likewise, performance optimization was not a goal, but significant steps were taken in the architecture to minimize the performance overhead normally associated with MAC. NSA and SCC then worked with the University of Utah's Flux research group to transfer the architecture to the Fluke research operating system [25]. During the transfer, what has become the *Flask* architecture was enhanced to provide better support for dynamic security policies.

The NSA created *Security-Enhanced Linux*, or *SELinux* for short, by integrating this enhanced architecture into the Linux operating system. It has been applied to the major subsystems of the Linux kernel, including the integration of mandatory access controls for operations on processes, files, and sockets. NAI Labs has since joined the effort and has implemented several additional kernel mandatory access controls, including controls for the *procfs* and *devpts* file systems. The MITRE Corporation and SCC have contributed to the development of some application security policies and have modified utility programs, but their contributions are not discussed further in this paper.

Using the flexibility of SELinux, it is possible to configure the system to support a wide variety of security policies. The system can support:

- separation policies that can enforce legal restrictions on data, establish well-defined user roles, or restrict access to classified data,
- containment policies useful for such things as restricting web server access to only authorized data and minimizing damage caused by viruses and other malicious code,
- integrity policies that are capable of protecting unauthorized modifications to data and applications, and
- invocation policies that can guarantee data is processed as required.

The flexibility of SELinux meets the goal of enabling many different models of security to be enforced with the same base system.

The NSA released the SELinux to make the technology available to a wider community and enable further research into secure operating systems. To help introduce the system in a more immediately useful form that helps demonstrate the added value of SELinux, NSA contracted NAI Labs to develop an example security policy configuration for the system designed to meet a number of common general-purpose security objectives. The example configuration greatly reduces the complexity of SELinux that would otherwise be present if building the policy specification from scratch were required. The example configuration released with the SELinux provides a customizable foundation with which a secure system can be built.

The remainder of this paper describes SELinux. It begins by providing an overview of the Flask architecture and its SELinux implementation in Section 2. The security mechanisms added to the system are then described in Section 3. The SELinux application programming interface (API) is discussed in Section 4. Section 5 describes the example security policy configuration created for the system. The performance overhead of the SELinux mechanisms is described in Section 6. Related work is discussed in Section 7.

2 Security Architecture

This section provides an overview of the Flask architecture and the SELinux implementation of the architecture. The Flask architecture provides flexible support for mandatory access control policies. In a system with mandatory access controls, a security label is assigned to each subject and object. All accesses from a subject to an object or between two subjects must be authorized by the policy based on these labels. The Flask architecture cleanly separates the definition of the policy logic from the enforcement mechanism. The security policy logic is encapsulated within a separate component of the operating system with well-defined interfaces for obtaining security policy decisions. This separate component is referred to as the *security server* due to its origins as a user-space server running on a microkernel. In the SELinux implementation, the security server is merely a kernel subsystem.

Components in the system that enforce the security policy are referred to as *object managers* in the Flask architecture. Object managers are modified to obtain security policy decisions from the security server and to apply these decisions to label and control access to their objects. In the SELinux implementation, the other kernel subsystems (e.g. process management, filesystem, socket IPC, System V IPC) are object managers. Application object managers can also be supported, such as a windowing system or a database management system.

The Flask architecture also provides an access vector cache (AVC) component that stores the access decision computations provided by the security server for subsequent use by the object managers. The AVC component also supports revocation of permissions, as described later in Section 2.4. An object manager may further reduce the cost of a permission check by storing references to the appropriate entry in the AVC with its objects. As a result, most permission checks can occur without even incurring the cost of an extra function call.

The remainder of this section further elaborates on the Flask architecture and its SELinux implementation. It begins by discussing how security labels are encapsulated in Flask. This section then discusses how Flask supports flexibility in labeling and access decisions. The ability of Flask to support policy changes is then described.

2.1 Encapsulation of Security Labels

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

The Flask architecture merely specifies the interfaces provided by the security server to the object managers. The implementation of the security server, including any policy language it may support, are not specified by the architecture. The SELinux example security server defines a security policy that is a combination of Type Enforcement (TE) [8], role-based access control (RBAC) [11], and optionally multi-level security (MLS) [7]. The example configuration for the TE and RBAC policy components is described in Section 5. The SELinux example security server defines a security context as containing a user identity, a role, a type, and optionally a MLS level or range. Roles are only relevant for processes, so file security contexts have a generic *object_r* role. The security server only provides SIDs for security contexts with legal combinations of user, role, type, and level or range. The individual attributes of the security context are not manipulated by the object managers.

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

ret = security_transition_sid(
        current->sid,
        dir->i_sid,
        SECCLASS_FILE,
        &sid);
```

Figure 1: Interface and example call to obtain a security label. The input parameters are the subject SID, the SID of a related object (e.g. the parent directory), and the class of the new object. The SID for the new object is returned as an output parameter.

The user identity attribute in the security context is independent of the ordinary Linux user identity attributes. Modifications to the Linux login program and cron daemon are provided to set this new user identity attribute appropriately for login sessions and user cron jobs. By using a separate user identity attribute, the SELinux mandatory access controls remain completely orthogonal to the existing Linux access controls. SELinux can enforce rigorous controls over changes in its user identity attribute without affecting compatibility with existing applications.

2.2 Flexibility in Labeling Decisions

When a Flask object manager requires a label for a new object, it consults the security server to obtain a labeling decision based on the label of the creating subject, the label of a related object, and the class of the new object. For program execution, the Flask process manager obtains the label for the transformed process based on the current label of the process and the label of the program executable. For file creation, the Flask file system object manager obtains the label for the new file based on the label of the creating process, the label of the parent directory, and the kind of file being created. The security server may compute the new label based on these inputs and may also use other external information. Figure 1 shows the security server's *security_transition_sid* interface for obtaining a label and an example call to this interface to obtain the label of a new file.

The SELinux example security server may be configured to automatically cause changes in the role or domain attributes of a process based on the role and domain of the process and the type of the program. By default, the role and domain of a process is not changed by program execution. The SELinux security server may also be configured to use specified types for new files based

```
int security_compute_av(
        security_id_t ssid,
        security_id_t tsid,
        security_class_t tclass,
        access_vector_t requested,
        access_vector_t *allowed,
        access_vector_t *decided,
        __u32 *seqno);
```

Figure 2: Interface for obtaining access decisions from the security server. The input parameters are a pair of SIDs, the class of the object, and the set of requested permissions. The pair of SIDs may be subject-to-object, subject-to-subject, or even object-to-object. The granted permissions are returned as output parameters.

on the domain of the process, the type of the parent directory, and the kind of file. A new file inherits the same type as its parent directory by default. For objects where there is only one relevant SID, object managers typically do not consult the security server. Instead, they merely use this SID as the SID for the new object. Pipes, file descriptions, and sockets inherit the SID of the creating process, and output messages inherit the SID of the sending socket.

2.3 Flexibility in Access Decisions

Object managers consult the AVC to check permissions based on a pair of labels and an object class, and the AVC obtains access decisions from the security server as needed. Figure 2 shows the security server's *security_compute_av* interface for obtaining access decisions. Figure 3 shows the AVC's *avc_has_perm_ref* interface for checking permissions and an example call to this interface to check bind permission to a socket.

Each object class has a set of associated permissions. These permission sets are represented by a bitmap called an *access vector* (*access_vector_t*). Flask defines a distinct permission for each service, and when a service accesses multiple objects, Flask defines a separate permission to control access to each object. For example, when a file is unlinked, Flask checks *remove_name* permission to the directory and *unlink* permission to the file.

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

2.4 Support for Policy Changes

The Flask AVC provides an interface to the security server for managing the cache as needed for policy changes. Sequence numbers are used to address the po-

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

ret = avc_has_perm_ref(
        current->sid,
        sk->sid, sk->sclass,
        SOCKET__BIND,
        &sk->avcr);
```

Figure 3: AVC interface and example call to check permissions. The input parameters are the same as for security_compute_av, except for the additional aeref parameter. On its first use, the aeref parameter is set to refer to the AVC entry used for the permission check, and on subsequent checks this reference is used to optimize the lookup. The reference is revalidated on each use to ensure its correctness.

tential interleaving of access decision computations and policy change notifications. When the AVC receives a policy change notification, it updates its own state and then invokes callback functions registered by the object managers to update any permissions retained in the state of the object managers. For example, permissions may be retained in the access rights in page tables or in the flags on an open file description. After updating the state of the object managers and the state of the AVC to conform to the policy change, the AVC notifies the security server that the transition to the new policy has been completed.

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

The SELinux example security server provides an interface for changing the security policy configuration at runtime. The *security_load_policy* call may be used to read a new policy configuration from a file. After loading the new policy configuration, the security server updates its SID mapping, invalidating any SIDs that are no longer authorized, and resets the AVC. Subsequent permission checks on processes and objects with invalid

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

Table 1: Permissions for the process object class.

SIDs always fail, preventing any further accesses by such processes and any further accesses to such objects. Support for automatically relabeling these processes and objects to a label that is accessible to administrators has not yet been implemented.

3 Security Mechanisms

This section describes the security mechanisms defined by the Flask architecture and the SELinux implementation of these mechanisms. It begins with a discussion of the mandatory access controls for process management. Mandatory access controls for file system objects are then described. This section concludes with a discussion of socket mandatory access controls.

3.1 Process Controls

Table 1 shows the permissions defined for the process management component. The process *execute* permission is used to control the ability of a process to execute from a given executable image. This permission is checked between the label of the transformed process and the label of the executable on every program execution. It is also checked when an ELF or script interpreter is executed, and when a file is memory-mapped with execute access (i.e. a shared library). This process *execute* permission is distinct from the file *execute* permission which is used to control the ability of a process to initiate the execution of a program.

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

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

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

The *ptrace* permission is used to control the ability of a process to trace another process. The *getsched*, *setsched*, *getsession*, *getpgid*, *setpgid*, *getcap*, and *setcap* permissions are used to control the ability of a process to observe or modify the corresponding attributes of another process.

In addition to the permissions listed in this table, SELinux provides an equivalent permission for each Linux capability. This allows the security policy to control the use of capabilities based on the SID of the process. SELinux could also be extended to provide a finer-grained replacement mechanism for capabilities. Such a mechanism was developed for one of SELinux's predecessors, the DTOS system [20]. This mechanism permitted privileges to be granted based on both the attributes of the process and the attributes of the relevant object, *e.g.* discretionary read override could be granted to a particular set of files. Since the mechanism obtained privilege decisions from the security server, management of privileges was centralized and verification that privileges were granted appropriately was straightforward.

3.2 File Controls

Table 2 lists the permissions for controlling access to open file description objects. Since open file descriptions may be inherited across *execve* or transferred through UNIX socket IPC, SELinux labels and controls open file descriptions. An open file description is labeled with the SID of its creating process, since its state is usu-

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

Table 2: Permissions for the open file description object class.

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

Table 3: Permissions for the file system object class.

ally treated as part of the private state of the process. It is important to distinguish between the label of an open file description and the label of the file it references. A read operation on a file changes the file offset in the open file description, so it may be necessary to prevent a process from reading a file using an open file description received or inherited from another process even though the process is allowed to directly open and read the file.

Permissions for controlling access to file systems are shown in Table 3. SELinux labels file systems and controls services that manipulate file systems, including calls for mounting and unmounting file systems, the *statfs* call and the file creation calls. SELinux controls the mounting of file systems through several permission checks. It requires that the process have *mounton* permission to the mount point directory and *mount* permission to the file system. It also requires that the *mountassociate* permission be granted between the root directory of the file system and the mount point directory.

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

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

SELinux currently only implements file labeling for the *ext2* file system. However, only the binding between on-disk inodes and PSIDs is filesystem-specific, so support for other local file system types can be easily added. For NFS file systems, a single label is currently used for all files mounted from a given NFS server. A design has been developed to provide complete file labeling and controls for NFS filesystems, but this design has not yet been implemented. SELinux also implements file labeling for the special *procfs* and *devpts* file systems based on the labels of the associated process, but these special file system types do not require the use of persistent label mappings.

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

Table 4 shows the permissions defined for controlling access to files, and Table 5 shows the additional permissions defined for controlling access to directories. SELinux defines a separate permission for each file and directory service. For example, SELinux defines an *append* permission for files in addition to the *write* permission, and it defines separate *add_name* and *remove_name* permissions for directories to support append-only files and directories. SELinux also defines a *reparent* permission for directories that controls whether the parent directory link can be changed by a *rename*.

SELinux provides control over each object affected by a file or directory service. For example, in addition to

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

Table 4: Permissions for the pipe and file object classes.

checking access to the parent directory, SELinux defines permissions for controlling access to the individual file itself for operations such as *stat*, *link*, *rename*, *unlink*, and *rmdir*.

3.3 Socket Controls

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

Since sockets are accessed through file descriptions, the socket object classes inherit the permissions defined for controlling access to the file object classes. Only a subset of these permissions are meaningful for sockets. Table 6 shows additional permissions that are specifically defined for controlling access to the socket object classes. The connection-oriented service provided by stream sockets requires several additional permissions, as shown in Table 7. Permissions for network interfaces and nodes are shown in Table 8.

Sockets effectively serve as communication proxies for processes in the SELinux control model. Consequently, sockets are labeled with the label of the creating process by default. A process may create and use a socket with a different label to perform socket IPC with

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

Table 5: Additional permissions for the directory object class.

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

Table 6: Additional permissions for the socket object classes.

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

Table 7: Additional permissions for the TCP and Unix stream socket object classes.

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

Table 8: Permissions for the network interface and node object classes.

a different source security label. A process may set up a listening socket so that server sockets created by connections are labeled with either a specified label or with the label of the connecting client socket to act as a server for multiple labels.

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

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

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

Support for communicating message labels across the network has not yet been implemented in SELinux. The Fluke implementation of the Flask architecture used IPSEC/ISAKMP both to label and protect messages, storing the labeling information in the IPSEC security association. During an ISAKMP negotiation, the appropriate security contexts are sent across the network and the peer obtains SIDs for these security contexts and stores them in its IPSEC security association. When messages are subsequently received that use the IPSEC security association, the messages are validated and then labeled with the SIDs from the association. Similar support will be provided in SELinux using the FreeSWAN [14] IPSEC implementation. Integrating FreeSWAN with the SELinux network mandatory access controls is the next major phase for SELinux development.

4 Application Programming Interface

Typically, the SELinux mandatory access controls operate transparently to applications and users. The labeling decisions of the Flask architecture provide appropriate default behaviors so that the existing Linux application programming interface (API) calls can be left unchanged. The mandatory access controls are only visible to applications and users upon access failures, in which case they return the normal Linux error codes (e.g. `EACCES`, `EPERM`, `ECONNREFUSED`, `ECONNRESET`) for such failures. In most cases, the potential for these same error conditions already existed with the ordinary Linux kernel, so most applications should handle these conditions. Only a few controls, such as the controls on individual *read* and *write* calls, can cause access failures where an access failure was not previously possible.

Although existing applications can be used unmodified, it is desirable to provide new API calls to allow modified and new applications to be developed that have some degree of awareness of the new security features. Each SELinux kernel subsystem provides a set of new API calls that extend existing API calls with additional parameters for SIDs. The process management subsystem provides calls to get the current and old SIDs of a process, and a call to execute a program with a specified SID. The filesystem subsystem provides calls to create files with particular SIDs, calls to obtain the SIDs of files and filesystems, and calls to change the SIDs of files and file systems. The socket IPC subsystem provides calls to create sockets and messages with particular SIDs, calls to obtain the SIDs of sockets and messages, and calls to specify the desired SID for peer sockets. The same set of controls used for the existing API calls are also applied to these extended API calls, with the only difference being the use of an application-provided SID rather than a default SID.

Applications that use these new calls need to be able to convert between SIDs and security contexts. Furthermore, it is desirable to allow applications to obtain security policy decisions from the security server so that security policies can be defined that control access to application abstractions. For example, a windowing system might be enhanced to provide labeling and separation of windows, with controlled cut-and-paste between windows, or a database system might be enhanced to provide labeling and separation of individual database records maintained in a single file. Such application policy enforcers would still be controlled by the kernel mandatory access controls but could further refine the granularity of protection provided by the kernel. To support such applications, the security server provides a set of new API calls that export its services for converting between SIDs and contexts and obtaining security policy decisions. A set of controls is defined for these new API calls to ensure that the policy can control the ability to use them. An application access vector cache library could easily be created based on the SELinux kernel access vector cache implementation to provide security decision caching for applications.

5 Security Policy Configuration

This section describes the example security policy configuration that has been developed for the Security-Enhanced Linux. At a high level, the goals of the example security policy configuration are to demonstrate the flexibility and security of the mandatory access controls and to provide a simple working system with minimal modifications to applications. The example security policy configuration consists of a combination of Role-Based Access Control (RBAC) [11] and Type Enforcement [8]. The configuration draws from the Domain and Type Enforcement (DTE) configuration described in [26], although it uses a different configuration language described in [16].

The example 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 program types can be used to enter each domain and the allowable transitions between domains. It also specifies automatic transitions between domains when certain program 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 user roles are initially set by the `login` program and can be changed by a `newrole` program similar to the `su` program.

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.

Figure 4 shows a portion of the policy configuration that allows the administrator domain (*sysadm_t*) to run the insmod program to insert kernel modules. The insmod program is labeled with the *insmod_exec_t* type and runs in the *insmod_t* domain. The first rule allows the *sysadm_t* domain to run the insmod program. The second rule allows the *sysadm_t* domain to transition to the *insmod_t* domain. The third rule allows the *insmod_t* domain to be entered by the insmod program and to execute code from this program. The fourth rule allows the *insmod_t* domain to inherit and use file descriptors from the *sysadm_t* domain. The fifth rule allows the *insmod_t* domain to use the `CAP_SYS_MODULE` capability. The last rule allows the *insmod_t* domain to send the `SIGCHLD` signal to *sysadm_t* when it exits.

```
allow sysadm_t insmod_exec_t:file x_file_perms;
allow sysadm_t insmod_t:process transition;
allow insmod_t insmod_exec_t:process { entrypoint execute };
allow insmod_t sysadm_t:fd inherit_fd_perms;
allow insmod_t self:capability sys_module;
allow insmod_t sysadm_t:process sigchld;
```

Figure 4: Configuration for running insmod.

From this small portion of the policy configuration, it is clear that the flexibility of the mandatory access controls also yields a corresponding increase in the complexity of managing the security policy. Creating and maintaining a configuration to meet a set of security requirements and verifying that the configuration is consistent with those requirements can be a challenging task. In order for SELinux to be widely deployed and used, a collection of base policy configurations must be developed to meet common sets of security requirements to allow its use by end users with no security expertise. Furthermore, higher-level configuration languages and policy analysis tools are needed to address these challenges.

The security policy configuration controls 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 configuration protects 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. As illustrated by the example in Figure 4, the configuration 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 integrity of system software, system configuration information and system logs is protected by the configuration. 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 configuration confines 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.

Privileged processes are protected 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 configuration ensures that the administrator role and domain cannot be 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. To provide confidentiality of secret authentication information, the policy configuration labels the shadow password file with its own type and restricts the ability to read this type to authorized programs such as `login` and `su`.

Ordinary user processes are prevented 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 configuration protects 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.

6 Performance

This section discusses the impact of the SELinux security mechanisms on the performance of the the Linux kernel. The set of benchmarks used was influenced by the *Linux Benchmarking HOWTO* [6]. Microbenchmark tests were performed to determine the performance overhead due to the SELinux changes for various low-level system operations. Macrobenchmark tests were performed to determine the impact of the SELinux changes on the performance of typical workloads.

Each test was performed with two different kernel configurations. The *base* kernel configuration corresponds to an unmodified Linux 2.4.2 kernel. This configuration was measured to provide the performance baseline for each benchmark. The *selinux* configuration corresponds to an enforcing Security-Enhanced Linux 2.4.2 kernel. The performance measurements of the *selinux* configuration can be compared against the baseline to determine the overhead imposed by the SELinux security mechanisms.

6.1 Microbenchmarks

The microbenchmark tests were drawn from the UnixBench 4.1.0 benchmark [21] and the lmbench 2 benchmark [18] suites. These microbenchmark tests were used to determine the performance overhead of the SELinux changes for various process, file, and socket low-level operations. These benchmarks were executed on a 333MHz Pentium II with 128M RAM. The lmbench network tests ran server programs on a 166MHz Pentium with 64MB RAM. Both the client and server machines ran the same kernel for the lmbench network benchmarks so that the results show the total cost of the SELinux overhead on both systems.

6.1.1 UnixBench The results for the UnixBench system microbenchmarks are shown in Table 9. The file copy benchmark measures the rate at which data can be transferred from one file to another, using various buffer sizes. For small buffer sizes, the system call overhead dominates the time to copy the file. The SELinux overhead consists of revalidating permissions for each read and write for the file copy. As the buffer size increases, the time to copy the file becomes dominated by the unaf-

Microbenchmark	Base	SELinux	Overhead
file copy 4KB	49.5	48.6	2%
file copy 1KB	40.4	38.6	5%
file copy 256B	23.0	21.0	10%
pipe	6.17	7.17	16%
pipe switching	12.7	15.0	18%
process creation	485	494	2%
execl	2480	2610	5%
shell scripts (8)	659	684	4%

Table 9: UnixBench system microbenchmarks. File copy throughput is in megabytes per second. The other UnixBench microbenchmarks are in microseconds per loop iteration (or milliseconds for the shell scripts benchmark). These results were converted into units that can be more easily compared with the lmbench results.

Microbenchmark	Base	SELinux	Overhead
null I/O	1.45	1.93	33%
stat	8.06	10.3	28%
open/close	11.0	14.0	27%
0KB create	22.0	26.0	18%
0KB delete	1.72	1.90	10%
fork	499	505	1%
execve	2730	2820	3%
sh	10K	11K	10%
pipe	12.5	14.0	12%
AF_UNIX	20.6	24.6	19%
UDP	310	356	15%
RPC/UDP	441	519	18%
TCP	389	425	9%
RPC/TCP	667	726	9%
TCP connect	675	738	9%

Table 10: lmbench microbenchmarks. Measurements are in microseconds. Measurements below the bar represent round-trip latency for various forms of IPC.

fected memory copying costs, so the SELinux overhead becomes negligible.

The pipe benchmark measures the number of times a process can write 512 bytes to a pipe and read them back per second. The pipe switching benchmark measures the number of times two processes can exchange an increasing integer through a pipe. The SELinux overhead consists of revalidating permissions for each read and write on the pipe.

The process creation test measures the number of times a process can fork and reap a child that immediately exits. The SELinux overhead consists of performing a permission check on each fork and wait operation. The execl benchmark measures the number of execl calls that can be performed per second. The SELinux overhead consists of computing the label for the transformed process and performing permission checks for searching the path, executing the program, and inheriting open file descriptions.

The shell scripts test measures the number of times per minute a process can start and reap a set of 8 concurrent copies of a shell script, where the shell script applies a series of transformations to a data file. The SELinux overhead consists of computing the label for processes for each program execution, computing the label for new files created by the scripts, and performing permission checks for the various process and file operations.

6.1.2 lmbench The results for the lmbench microbenchmarks are shown in Table 10. The null I/O benchmark measures the average of the times for a one-byte read from */dev/zero* and a one-byte write to */dev/null*. The SELinux overhead consists of revalidating permissions on each read and write.

The stat benchmarks measures the time to invoke the *stat* system call on a temporary file. The SELinux overhead consists of performing permission checks for searching the path and obtaining the file attributes. The open/close test measures the time to open a temporary file for reading and immediately close it. The SELinux overhead consists of performing permission checks for searching the path and opening the file with read access.

The 0K create and 0k delete benchmarks measure the time required to create and delete a zero-length file. For the 0K create, the SELinux overhead consists of computing the label for the new file and performing permission checks for searching the path, modifying the directory, and creating the file. The SELinux overhead for the 0K delete consists of performing permission checks for searching the path, modifying the directory, and unlinking the file.

The fork, execve, and sh benchmarks measure three increasingly expensive forms of process creation: fork and exit, fork and execve, and fork and execlp of the shell with the new program as a command to the shell. For the fork benchmark, the SELinux overhead consists of permission checks on fork and wait, as with the UnixBench process creation benchmark. For the execve benchmark, the SELinux overhead consists of the fork overhead plus the label computation and permission checks associated with program execution, as with the UnixBench execl benchmark. For the sh benchmark, this overhead is further increased by the additional layer of process creation, program execution, and path searching by the shell.

The remaining lmbench tests measure round-trip latency in microseconds for various forms of interprocess communication between a pair of processes. The lmbench bandwidth benchmark results are omitted since they did not show any significant difference between the *base* and *selinux* configurations, as expected.

The SELinux overhead on the pipe benchmark consists of revalidating permissions on each read and write, as with the UnixBench pipe switching benchmark. For the AF_UNIX benchmark, the SELinux overhead consists of checking permission to each socket and revalidating the permissions for the connection between the sockets on each send and receive. For each of the networking benchmarks, the SELinux overhead includes checking permission to each socket, host, and network interface for each packet. The overhead for the UDP and RPC/UDP benchmarks also includes checking permission between the socket pair on each send and receive. For the TCP and RPC/TCP benchmarks, SELinux revalidates the permissions granted during connection establishment between the socket pair on each send and receive. The SELinux overhead for the TCP connection benchmark includes the permission checks between the socket pair for the connection on connect and accept.

6.1.3 Conclusion

Although the percentage overhead for some of the microbenchmark results is large, the real difference in absolute times is typically quite small and becomes insignificant for macro operations, as shown by the results in Section 6.2. Furthermore, these results must be viewed as an upper bound on the performance overhead, since neither the AVC nor the security server implementation have been optimized. Other known areas where the performance could be improved include making better use of AVC entry references and improving the AVC locking scheme.

6.2 Macrobenchmarks

The first macrobenchmark consisted of compiling the Linux 2.4.2 kernel sources, since this involves significant file system activity and is representative of a workload experienced commonly by Linux users. The second macrobenchmark was the WebStone 2.5 benchmark for web servers [19], which is representative of a typical workload for a web server.

For the kernel compilation macrobenchmark, the time to execute "make" was measured. The 2.4.2 kernel sources were configured with the default options, and a "make dep" was done prior to the testing. Three kernel compilations were performed, each immediately after a reboot into single-user mode, and the results were averaged. This benchmark was executed on a 333MHz Pentium II with 128M RAM.

For the WebStone macrobenchmark, one hundred 10-minute trials were run with 32 web clients requesting the standard WebStone file set. A Sun Ultra 5 running SunOS 5.6 with 128M RAM was used as the test controller and client machine. This machine was directly connected using a 10Mbit Ethernet crossover cable to a 133MHz Pentium with 64M RAM running the Apache web server provided with RedHat 6.1.

Table 11 displays the results of the macrobenchmarks.

	Base	SELinux	Overhead
elapsed	11:14	11:15	0%
system	00:49	00:51	4%
latency	0.56	0.56	0%
throughput	8.29	8.28	0%

Table 11: Macrobenchmark results. The elapsed and system times for a "time make" on the Linux 2.4.2 kernel sources are shown in minutes and seconds. The latency in seconds and throughput in MBits per second are shown for the WebStone benchmark.

There was no significant change in the total elapsed time, and there was only a 4% increase in the system time for a kernel compilation. There was no significant change in either the latency or the throughput measurements for WebStone. At the macro level, there appears to be little noticeable difference.

7 Related Work

The project that is most similar to SELinux is the Rule Set Based Access Control (RSBAC) [22] for Linux project. RSBAC is based on the Generalized Framework for Access Control (GFAC) [4]. Like the Flask architecture, the GFAC separates policy from enforcement and can support a variety of security policies. RSBAC provides a Role Compatibility policy module that is very similar to the SELinux Type Enforcement policy module.

However, RSBAC also differs from SELinux in a number of ways. The GFAC does not specifically address the issue of atomic policy changes, so RSBAC lacks the SELinux support for dynamic security policies. Since the GFAC places the responsibility for managing security labels in its Access Control Information (ACI) module, RSBAC does not provide policy-independent data types for security labels. The RSBAC Access Decision Facility (ADF) depends on kernel-specific data structures, and RSBAC does not provide a security decision cache mechanism, because the RSBAC ADF was directly implemented as a kernel subsystem. In contrast, since SELinux's predecessor systems implemented the security server as a user-space server running on a microkernel, the SELinux security server is cleanly decoupled from the kernel and SELinux provides the access vector cache.

Since RSBAC was not designed with security-aware applications and application policy enforcers in mind, it lacks equivalents for the extended API calls and new API calls of SELinux, only providing calls for setting and getting attributes of existing subjects and objects. RSBAC uses the Linux real user identity attribute for its decisions and must control changes to this attribute, so it is not completely orthogonal to the existing Linux access controls. Finally, RSBAC lacks a number of the controls provided by SELinux for each of the kernel subsystems.

Type Enforcement [8] (TE) and Domain and Type Enforcement (DTE) [5] have a number of similarities to SELinux, since SELinux provides a generalization of TE in its example security server. Two projects are integrating DTE into Linux [15, 1]. SELinux was designed to provide flexible support for a variety of policy models, while DTE was only designed to implement an enhanced form of TE. DTE is distinguished from traditional TE by the DTE Language (DTEL) for expressing access control configurations and by an implicit typing mechanism based on the directory hierarchy for labeling files. The SELinux TE policy module likewise has a configuration language for expressing access control rules. SELinux stores file labels explicitly, but allows labels to be managed using a higher-level specification based on pathname regular expressions. NAI Labs' DTE prototype also provided labeling and controls for NFS and was integrated with IPSEC.

The TrustedBSD project is developing a variety of trusted operating system features, including mandatory access controls, for FreeBSD [27]. SELinux differs from TrustedBSD in that SELinux is a more mature system, that it addresses only mandatory access controls, and that it uses a flexible mandatory access control architecture rather than hardcoded policies. The TrustedBSD project plans to migrate to a more flexible mandatory access control architecture in the future [28].

The Medusa DS9 [3] project is similar to SELinux at a high level in that it is also developing a kernel access control architecture that separates policy from enforcement. However, Medusa is very different in its specifics. In Medusa, the kernel consults a user-space authorization server for access decisions. The Medusa access controls are primarily based on labeling subjects and objects with sets of virtual spaces to which they belong and defining what virtual spaces can be seen, read, and written by each subject. The authorization server can also require explicit authorization in addition to the virtual space checking, in which case it can apply other kinds of policy logic and can even override the ordinary Linux access controls. Medusa DS9 also provides support for system call interception by the authorization server and for forcing a process to execute code provided by the authorization server.

The Linux Intrusion Detection System (LIDS) [2] provides a set of additional security features for Linux. It supports administratively-defined access control lists for files that identify subjects based on their program. Like Medusa, LIDS can control the ability to see files and processes in directory listings. LIDS also supports defining capability sets for programs, preventing certain processes from being killed, sending security alerts on access failures, and detecting port scans.

The LOMAC [13] project has implemented a form of mandatory access control based on the Low Water-Mark model in a Linux loadable kernel module. LOMAC was not designed to provide flexibility in its support for security policies; instead, it focuses on providing useful integrity protection without any site-specific configuration, regardless of the software and users present on a system. It should be possible to implement the Low Water-Mark model in SELinux as a particular policy module.

8 Summary

This paper explains the need for mandatory access control (MAC) in mainstream operating systems and presents the NSA's implementation of a flexible MAC architecture called Flask in the Security-Enhanced Linux (SELinux) prototype. The paper explains how the Flask architecture separates policy from enforcement and provides the necessary interfaces and infrastructure for flexible policy decisions and policy changes. It describes the fine-grained labeling and controls provided by SELinux for kernel objects and services. The paper explains how existing Linux applications can run unchanged on the SELinux kernel, and it describes the support for security-aware applications. The paper shows how the SELinux controls can be applied to meet real security objectives by describing the example security policy configuration. It demonstrates that the performance overhead of the SELinux controls is minimal. Finally, the paper highlights the differences between SELinux and related systems.

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 and for his assistance in porting the kernel modifications to the 2.4 kernel. We thank Anthony Colatrella and Timothy Fraser for assisting with the performance benchmarking and analysis. We also thank Ted Faber, Timothy Fraser and the anonymous reviewers for reviewing earlier drafts of this paper.

References

[1] Configurable Access Control Effort. http://research–cistw.saic.com/cace.

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

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

[4] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. A Generalized Framework for Access Control: An Informal Description. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 135–143, Oct. 1990.

[5] 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.

[6] A. D. Balsa. Linux Benchmarking HOWTO, Aug. 1997. http://-www.linuxdoc.org/HOWTO/Benchmarking-HOWTO.html.

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[14] J. Gilmore. FreeSWAN. http://www.freeswan.org.

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

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

[17] 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.

[18] L. McVoy and C. Staelin. lmbench 2. http://www.bitmover.com/-lmbench.

[19] MindCraft. Webstone. http://www.mindcraft.com/webstone.

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

[21] D. C. Niemi. Unixbench 4.1.0. http://www.tux.org/pub/tux/-niemi/unixbench.

[22] 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.

[23] Secure Computing Corp. DTOS Formal Security Policy Model. DTOS CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, Sept. 1996. http://www.securecomputing.com/randt/HTML/-dtos.html.

[24] Secure Computing Corp. DTOS Generalized Security Policy Specification. DTOS CDRL A019, 2675 Long Lake Rd, Roseville, MN 55113, June 1997.

[25] 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.

[26] 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.

[27] R. Watson. Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD. In *Proceedings of the 2000 BSD Conference and Expo*, Oct. 2000.

[28] R. Watson. Robert Watson on FreeBSD and TrustedBSD, Jan. 2001. http://slashdot.org/interviews/01/01/18/1251257.shtml.