Security Enhanced BSD

Chris Vance, Robert Watson
Network Associates Laboratories *
15204 Omega Drive, Suit 300
Rockville, MD 20850
cvance@nai.com, rwatson@nai.com

July 9, 2003

*This work was supported in part by DARPA/SPAWAR contract N66001-01-C-8035.

Contents

1 Introduction

Network Associates Laboratories has completed an initial port of the Flask security architecture[1] and other components of Security Enhanced Linux (SELinux)[2] to the FreeBSD[3] operating system. This project, called Security Enhanced BSD (SEBSD), started with the TrustedBSD MAC framework and integrated the Flask access vector cache and security server to make policy decisions. Then, support was added to the kernel to manage security fields and enforce permissions on files and processes.

To demonstrate the resulting kernel functionality, a policy compiler and file system label management tools were ported. Also, modifications to login, ls, and the ps program were integrated into the corresponding FreeBSD programs. This paper discusses the TrustedBSD MAC framework, label management, access control checks, and differences between SEBSD and SELinux.

2 Background

The introduction of new access control security features into operating systems is an expensive process, both from the perspective of development, and in terms of long-term maintenance. A variety of approaches for security extension exist, but all have substantial problems, ranging from specific concerns over technical correctness to high maintenance costs. Many operating system security extensions rely on modifications to the kernel to operate, preventing mandatory protections from being bypassed. However, this is often done at the expense of flexibility. For these reasons, both the Linux and FreeBSD open source operating system projects began the development of generic, extensible security frameworks to help reduce these long-term costs and to help foster research into better operating system access controls.

2.1 TrustedBSD Mandatory Access Control Framework

Network Associates Laboratories and the TrustedBSD Project have implemented an extensible and modular kernel access control framework permitting new access control policies to be introduced into the FreeBSD kernel[4, 5, 6, 7, 8]. The TrustedBSD Mandatory Access Control (MAC) Framework addresses many of the challenges associated with introducing new access control services in operating

system kernels by abstracting common infrastructure services from the policies, reducing the cost and complexity of policy authoring. This includes providing policy-independent label storage in kernel objects, and persistent storage of labels using file system extended attributes. The TrustedBSD MAC Framework composes results from simultaneously loaded access control policies in a predictable and reliable manner, permitting appropriately crafted policies to be used in concert.

The MAC framework augments the FreeBSD kernel to provide common labeling infrastructure along with a set of entry points to intercept operations on labeled objects. The framework supports labels on file systems, processes, IPC, and network stack elements. Each registered policy may reserve space for security labels and implement policy-specific behavior governing label content and use. Labels follow the kernel object life cycle and are initialized, allocated, and destroyed along with their object. Access control entry points accept information about the action being performed, invoke each registered policy, and compose the results into a success or failure.

The following table lists the FreeBSD kernel objects that contain MAC labels:

Structure	Description
struct ucred	Process credential
struct file	File descriptor
struct vnode	VFS node
struct socket	BSD IPC socket
struct pipe	IPC pipe
struct mbuf	In-flight datagram
struct mount	File system mount
struct ifnet	Network interface
struct devfs_dirent	Devfs entry
struct ipq	IP fragment queue
struct bpf_desc	BPF packet sniff device

On-going work with the MAC framework is designed to increase the scope of the access control entry points. Future versions will include support for the System V IPC kernel subsystem as well as better support for file system mount points; other kernel subsystems will be examined and the access control entry points will be refined as necessary. Network Associates Laboratories has also begun work to port the TrustedBSD MAC framework and the SEBSD module to the Darwin kernel; this will likely result in additional changes to the MAC framework to facilitate cross-platform development.

2.2 Linux Security Modules Framework

The Linux Security Modules (LSM) project was primarily developed by WireX and Network Associates Laboratories and seeks to incorporate a general security framework into the Linux kernel. LSM is a joint development effort by several projects, including Immunix, SELinux, and Janus, and several individuals, including Greg Kroah-Hartman and James Morris, to develop a Linux kernel patch that implements this framework.[9]

While LSM was originally developed as a set of patches that may be applied to the Linux kernel distribution, much of the security framework is now present in the currently distributed Linux development kernel (2.5.x series kernels). The LSM framework is primarily focused on supporting access control modules, but may be extended to support other security needs such as auditing. The LSM kernel patch moved most of the capabilities logic into an optional security module, with the system defaulting to the traditional superuser logic.

Much like the TrustedBSD MAC framework, LSM added security fields to kernel data structures and inserted calls to hook functions at critical points in the kernel code to manage the security fields and to perform access control. It also added functions for registering and de-registering security modules and ongoing work will provide a generic set of user-space interfaces to set and retrieve labels on kernel objects.

Security fields, that may be used to store labels or any other state information, were added to the following kernel data structures:

Structure	Description
struct task	Process label
struct linux_binprm	Binary handler
struct super_block	File system mount point
struct inode	File node (also sockets)
struct file	File handle
struct sk_buff	Network message buffer
struct net_device	Network interface
struct kern_ipc_perm	System V IPC object
struct msg_msg	System V message

By providing security fields for these structures and by providing appropriate operational hooks, an LSM module can provide access control over processes, pro-

grams, file systems, pipes, files, sockets, packets, network devices, and System V IPC objects.

SELinux was originally developed as a set of patches, directly modifying the Linux kernel, while more recent version use the LSM framework. Because Linux, LSM, SELinux are all still under development, various versions of SELinux are slightly different. The Linux kernel is developed with two primary branches, a stable branch and a current development branch. As development continues with the Linux kernel, the stable and current branches tend to diverge. While much of the LSM prototype has been included in the current Linux kernel branch, it is not complete; the LSM project still maintains patches to incorporate the remaining features. Since little of the framework is included in the stable kernel branch, the LSM project maintains a complete patch.

2.3 Framework Comparison

While the implementation details differ, both the Linux and FreeBSD frameworks seek to solve the same basic problems. Both frameworks permit access control modules to be dynamically inserted into an otherwise standard system, either at boot time or after boot, possibly in response to an environmental change. By itself, neither framework increases the security of the system; they merely provide the infrastructure necessary to support security modules. Both projects considered modularity and policy flexibility to be critical to the adoption of their respective security framework by the kernel developers; neither operating system wanted to be tied to a single security model or implementation.

Furthermore, both LSM and the TrustedBSD MAC framework operate transparently to existing users and applications. The results of access control decisions are only visible to applications upon failure, and in most cases, the kernel services will return appropriate error codes that applications should expect from an unmodified kernel. However, it is possible that security modules will return an access failure where not previously expected; this may cause unexpected side-effects in user space applications that are not security-aware.

While both frameworks had similar goals, the resulting frameworks were largely shaped by the requirements of the user community. In the case of LSM, any design that was too intrusive was unlikely to be accepted by the Linux kernel developers, so often sacrifices had to be made in order for the framework to be included in the kernel. On the other hand, the TrustedBSD MAC was designed with a goal of integrating tightly with the kernel locking and threading mechanisms to provide

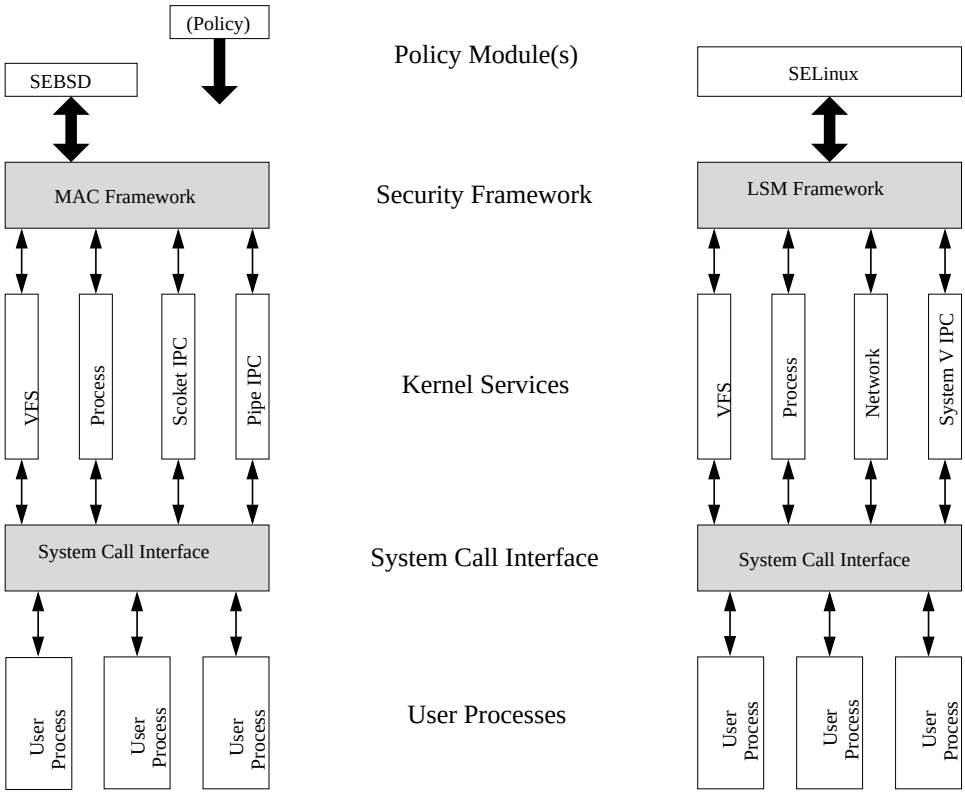

Figure 1: High-level view of the Frameworks

correctness and high performance on multi-cpu machines. While the LSM project attempted to minimize the changes to the base kernel, the TrustedBSD MAC framework was not developed under this restriction, and was able to restructure existing kernel code at places where access control decisions needed to be made. The MAC Framework guarantees that sufficient locks will be held in order to access important elements passed as arguments to access control entry point functions. Likewise, the MAC Framework provides well-defined locking semantics for object labels, often using existing object locks. Often, the locking semantics permit atomic checks of both labels and existing object properties without incurring additional locking overhead.

Policy composition is integral to the MAC Framework, rather than leaving composition up to the module writers, as LSM does. LSM chose to allow maximum flexibility by creating a truly generic framework that provides all the necessary

hooks and label management tools, but enforces no semantics on how they must be used. So, while there is no built-in support for module composition, all the necessary hooks are present to do so. When multiple policies are loaded into the MAC Framework, their access control decisions are usefully composed in a way that the results are well-defined. However, this composition is fully controlled and enforced by the framework, not by the policy developers.

To further support policy composition, the TrustedBSD MAC framework also provides policy neutral interfaces and user space tools. The MAC framework provides a policy-agnostic label management API to provide access to and management of file and process labels. Several FreeBSD common utility programs have been made label-aware (but policy-agnostic), such as ps, ls, and login. The TrustedBSD MAC framework has investigated techniques to provide policy-independent support for login and other applications with more complex labeling requirements, such as label transitions at program execution time. However, not all of these techniques are sufficient for SEBSD. The MAC framework must be expanded to include better support for SEBSD-specific features, while maintaining policy flexibility, ease of management, and configuration.

Both frameworks allow file system labels to be backed to persistent storage. While neither framework enforces the semantics of particular persistent backing mechanism, both Linux and FreeBSD can support both extended attributes and custom label backing stores. The Linux extended attribute kernel support is relatively new and untested, with little user space management tools. Historically, SELinux used it's own persistent file label store, rather than any provided by the file system. It is expected that support will improve as the extended attribute system matures. FreeBSD's UFS2 file system provides robust support for extended attributes, including centralized cache management for persistent file labels, as well as transaction-like support for consistency when applying labeling changes from compound operations across multiple policies.

The TrustedBSD MAC framework provides both label management entry points and access control entry points. The access control entry points always pass as parameters all information that may safely be used by policy developers. These parameters include the extracted user and process credentials, when available. Having the Framework pass in explicit label pointers reduces binary and source compatibility issues associated with changes to the base system structures. This may lower development costs and improve long term maintainability. In addition, to support high performance reliable operation on multi-CPU systems the explicit credential is also used by FreeBSD to permit deferred activity on behalf of a subject (i.e. NFS write-behind, ktrace to a disk file, etc.). The LSM framework typically assumes

that object labels may be extracted from the global current process context. This has caused some problems with the SELinux network labeling implementation, as the current process is not always available or correct.

As an example, the TrustedBSD MAC framework entry point for performing access control checks for the swapon system event is structured as follows:

```
int mpo_check_system_swapon(struct ucred *cred,
    struct vnode *vp, struct label *label);
```

Whereas the LSM framework provides the following hook:

```
int security_swapon(struct swap_info_struct * swap);
```

The TrustedBSD MAC framework provides the associated user credentials (and corresponding process label), whereas the LSM hook relies upon the module developer to extract this information from `current` (the current process context).

3 SELinux

NSA Security-Enhanced Linux (SELinux) is an implementation of a flexible and fine-grained mandatory access control (MAC) architecture called Flask in the Linux kernel[10, 1]. SELinux can enforce an administratively-defined security policy over all processes and objects in the system, basing decisions on labels containing a variety of security-relevant information. The architecture provides flexibility by cleanly separating the policy decision-making logic from the policy enforcement logic. The policy decision-making logic is encapsulated within a single component known as the security server with a general security interface. The policy enforcement logic is implemented using the interfaces specified by the LSM framework. A wide range of security models can be implemented as security servers without requiring changes to any other component of the system.

3.1 Flask

SELinux is based on the Flask security architecture for flexible non-discretionary access controls. The Flask security architecture specifies well defined interfaces

to provide a clean separation between policy enforcement and policy interpretation. The Flask security architecture also includes an access vector cache (AVC) component to help minimize the performance overhead from the access control computation. While policy enforcement code is largely system specific, policy interpretation and access control decision making code is platform independent. The policy enforcement code is tightly integrated into the kernel services it protects, and uses the Flask security server APIs (and the AVC) to obtain security policy decisions.

The Flask security architecture provides two policy-independent data types: the security context (context) and the security identifier (SID). The security context is a string representation of a policy-specific security label. The SID is a local integer identifier that may be used as a run-time handle to identify specific security context. The security server will maintain a set of security classes that identify the type of object being protected; each security class has an associated set of permissions for controlling access to the object. This associated set of permissions is represented as a bitmap called an access vector.

A Flask object manager binds SIDs to active kernel objects, and uses these SIDs as context during access control checks. The policy enforcement provides a source context, a target context, a security class, and an access vector to the AVC to determine access to an object. Likewise, when an object manager wishes to label a newly created object, it will consult the security server to obtain a label.

3.2 Modified Programs

The SELinux distribution also includes support programs and modified user space applications. The support programs include a policy compiler, a file system labeling tool, role management tools, and policy querying tools. Several user space applications were modified to make them policy-aware. In particular, SELinux distributes modified versions of GNU process and file utilities, log rotation programs, the system login program, the openssh server, the tar program, and vixie cron. The changes were primarily made to allow the programs to retrieve and display SELinux labeling information, in some cases, allowing programs to maintain existing labels as files are modified. Others programs (login, sshd, cron) were modified to support execution time modification of process labels.

4 SEBSD

This section provides an overview of the SEBSD security module architecture. Much like the SELinux module, TrustedBSD MAC policies are built as loadable kernel modules, relying on the FreeBSD module facilities for linking and loading.

4.1 Distribution

The source code for SEBSD is distributed as a stand-alone kernel module that may be linked against FreeBSD 5.x. Typically, it is desirable to include the SEBSD sources directly in the kernel source tree, in the `sys/security/sebsd` directory. The SEBSD implementation consists of the same major components as the LSM-based SELinux implementation: the security server, the access vector cache (AVC), new system call implementations, and the entry point function implementations. The exception is the fifth element of the SELinux architecture, the SELinux-specific persistent label mapping, it was not ported to FreeBSD, since the UFS and UFS2 file systems supported the use of native extended attributes and the TrustedBSD MAC Framework provides integrated support for handling extended attributes. Otherwise, SEBSD literally reuses (almost verbatim) the Flask AVC, the security server components, the policy compiler, and even much of the policy itself.

The policy configuration used by SEBSD is roughly the same as that provided by SELinux. For the most part, path names were changed to reflect a FreeBSD installation. It was not that difficult to configure the policy to support a base configuration of FreeBSD, allow it to boot in enforcing mode, and permit user and administrator login.

4.2 Porting Flask

The AVC and security server were only modified in ways necessary to port them to the FreeBSD operating system. Compatibility with the SELinux components was maintained when possible. Changes were necessary to:

- Replace memory allocations and deallocations with a generic wrapper function that implements FreeBSD kernel-specific operations.
- Allow the binary policy file to be accessed from within the kernel. The original LSM-based SELinux distribution handled policy initialization in this

manner, but has since migrated toward a user-space policy loading mechanism. It is expected that SEBSD will follow the lead from SELinux and will be modified to no longer directly look up or read the binary policy file from inside the kernel; the policy will be opened and read from user space and passed as a memory mapped data pointer to the kernel.

- Locking primitives were converted to FreeBSD equivalents.
- The AVC and the Security Server were separated into two corresponding sub-directories (`avc` and `ss`) within the SEBSD source hierarchy; this facilitated development and reduced inter-dependencies. Only FreeBSD-specific code resides at the top level of the source hierarchy.
- Audit information was updated to correctly report details of FreeBSD kernel data structures. Since full or partial path names are not readily available in the FreeBSD kernel, file system and file identifiers (`fsid` and `vnode`) are currently reported by avc_audit. The unavailablity of complete path names is a property of the VFS model employed by the FreeBSD kernel. While path names are frequently available in the Linux kernel, it is not always possible to reconstruct the complete path name.
- The user space interfaces to the SEBSD module were re-implemented in a generic manner so that they could be made available to all MAC modules. This led to the development of interfaces that use policy-independent textual representation for module names and labels. This provided the opportunity to avoid exposing SIDs outside the kernel, instead passing contexts as string based identifiers. SELinux later adopted this approach.

Other than the changes above, the major components of the Flask architecture remain functionally equivalent on FreeBSD and Linux.

4.3 SEBSD Module Initialization

The FreeBSD module interfaces allow policy modules to be linked into the kernel at build time, loaded prior to the kernel starting at boot time, or loaded at run-time. Since SEBSD requires ubiquitous access to all system objects, it must be present from system inception; the SEBSD module must either be linked directly into the kernel, or it may be built as a separate dynamic kernel module and configured to load prior to kernel execution.

The MAC Framework is initialized early in the boot process, shortly after basic kernel primitives are initialized (memory allocation, system console, and locking primitives), but prior to probing devices and starting any kernel or user processes. Once the framework is initialized, it will allow policy registration. Policy modules built directly into the kernel will be registered at this time. Likewise, modules loaded by the boot loader prior to boot will be registered at this time.

When policies register with the MAC Framework, they provide a number of properties that are used by the framework to properly identify and configure the module. Shown below are the properties set by the SEBSD module.

Property	Value
Module Name	sebsd
Full Name	NSA/NAI Labs Security Enhanced BSD
Uses Labels	yes
Flags	MPC_LOADTIME_FLAG_NOTLATE

In addition to setting the name for the module, the properties request storage for kernel object labels, and set a flag to indicate that the policy module must be loaded and initialized early in the boot process. The `NOTLATE` flag also means that attempts to register the module after the system boot will fail.

4.4 User Interfaces

SEBSD uses both the system control (sysctl) and system call interfaces to allow user space processes to access the state of the SEBSD module and to query the security server. Library functions, located in `libsebsd` wrap the sysctls and system calls, providing an API that is identical to that provided by SELinux. The following list of SELinux APIs are supported:

```
int sebsd_enabled(void);

int sebsd_enforcing(void);

int sebsd_load_policy(const char *path);

int get_ordered_context_list(const char *user_name,
    const char *from_context, char ***ordered_list,
```

```
    size_t *length);

int get_default_context(const char *username,
    const char *from_context, char **default_context);

int query_user_context(pam_handle_t *pamh,
    char **ordered_context_list, size_t length,
    char **retcontext);

security_class_t string_to_security_class(const char *s);

int security_get_user_contexts(const char *fromcontext,
    const char *username, char ***retcontexts,
    size_t *ncontexts);

int security_change_context(const char *domain,
    const char *ocontext, security_class_t oclass,
    char **newcontext);

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

SEBSD chose not to export SIDs from the kernel; user space applications will only have access to context strings. The TrustedBSD MAC framework developed generic string-based label management facilities that are compatible with all security policies. By requiring textual label representations in user space, TrustedBSD was free to develop applications conforming to a single standard, and user tools may be both label-aware and policy-agnostic. The SELinux project seems to be moving in this direction as well, so ultimately this design will allow the SELinux and SEBSD interfaces to converge.

4.4.1 System Controls

SEBSD uses five system controls (sysctls) to maintain module state information and to query the security server. The sysctls are wrapped by library functions in `libsebsd`. The sysctls for displaying module state information may be queried or updated with the normal FreeBSD administrative tools such as the `sysctl` command.

System Control	Description
security.mac.sebsd.enforcing	Display state of the enforcement of policy; allows modification to enable/disable enforcement
security.mac.sebsd.sids	List SIDS in active use by the security server
security.mac.sebsd.user_sids	Lists the SIDs currently available for transition to by a given context
security.mac.sebsd.change_sid	Report the SID to relabel to given input source context, target context, and a security class.
security.mac.sebsd.compute_av	Compute access vectors given input source and target contexts, security class, and access vector

4.4.2 System Calls

The TrustedBSD MAC framework includes an entry point function for a multiplexed system call. SEBSD is currently using this interface to support the reloading of security policy after boot time. It is anticipated that this interface will be converted to a sysctl and that no SEBSD-specific system calls will be required. There are three system calls currently defined; they are listed in the following table:

System Call	Description
int sebsd_enforcing()	Boolean: is the policy being enforced
int sebsd_avc_toggle()	Toggle the state of the enforcing flag
int sebsd_load_policy(cont char *path)	Load the specified policy file

4.5 Label Management Tools

The TrustedBSD MAC Framework provides a number of policy-agnostic interfaces for policy-aware applications; these interfaces are available in the standard C library. The framework provides interfaces to get and set labels on file system objects (vnodes), sockets, pipes, and network interfaces. These generic labeling services are used to make several basic system binaries policy-aware. The FreeBSD `ls` and `ps` utilities were modified to report file and process labels, and the `ifconfig` tool was modified to support network device labels. In addition, several new label management tools are provided:

Program	Description
getfmac	Retrieve file label
setfmac	Set file label
getpmac	Retrieve process label
setpmac	Set process label
setfsmac	Sets labels on the specified file system hierarchy

4.6 SEBSD User Space Applications

The SELinux policy compiler and initial file system labeling tools were ported to the FreeBSD operating system for use with SEBSD; these tools were renamed to `sebsd_checkpolicy` and `sebsd_setfiles` respectively. The policy compiler is essentially unchanged. The file system labeling tool was re-written to use generic file hierarchy traversal (fts) routines instead of the Linux nftw routines. This tool was also used as the basis for a generic file labeling tool, called `setfsmac`, which is able to apply policy-agnostic labels based on file specifications.

Since SELinux and SEBSD only permit process label transitions at program execution time, the FreeBSD login program was modified to use the `execve_secure` system call to permit new login shells to operate in the proper domain. To show how this may be done for other applications, the cron daemon was also modified to permit cron jobs to execute with the correct label. These changes have not yet been adopted by the TrustedBSD MAC framework, so they are SEBSD-specific applications. In continuing work, the MAC framework will design a policy-agnostic interface for login, providing compatible support.

5 SEBSD Label Management

5.1 SEBSD Labels

SEBSD maintains per-object labels on processes, pipes, files, file descriptors, and file systems. The labels contain information that SEBSD uses to make access control decisions. Each of the object-specific label structures are defined in `sebsd_labels.h`.

5.1.1 Process Labels

The `task_security_struct`, defined in `sebsd_labels.h`, contains security information for system and kernel processes. The MAC framework stores these labels in the process credential structure. The structure is defined as follows:

```
struct task_security_struct {
        security_id_t osid;
        security_id_t sid;
        avc_entry_ref_t avcr;
};
```

Field	Description
osid	SID prior to the last execve
sid	SID for the process
avcr	AVC entry reference

5.1.2 Mount Labels

The MAC framework maintains two separate labels for the kernel mount structure; SEBSD uses the `mount_security_struct` to label the file system mount point itself; this label is intended to be used to authorize mount, unmount, and stat calls. The second structure, the `mount_fs_security_struct`, is used as the default label for objects in the file system, when the file system does not support persistent file labels. As the TrustedBSD MAC framework matures, this may be controlled from user space with a mount option. This would allow the system administrator to specify an initial label at mount time. Due to locking concerns, neither label can be changed at run-time.

```
struct mount_security_struct {
        security_id_t sid;
        unsigned char uses_psids;
        unsigned char uses_task;
        unsigned char uses_genfs;
        unsigned char proc;
        unsigned char uses_trans;
};
```

```
struct mount_fs_security_struct {
        security_id_t sid;
};
```

Field	Description
sid	SID for the mount
uses_psids	Flag: this file system supports persistent SIDs
uses_task	Flag: use creating task SID for vnodes
uses_genfs	Flag: use `security_genfs_sid` for vnodes
uses_trans	Flag: whether to call `security_transition_sid`
proc	Flag: whether to call `procfs_set_sid`

5.1.3 File and Pipe Labels

The `vnode_security_struct`, contains security information for vnodes and represent objects within a file system. This label structure is also used to label pipe objects.

```
struct vnode_security_struct {
        security_id_t task_sid;
        security_id_t sid;
        security_class_t sclass;
        avc_entry_ref_t avcr;
};
```

Field	Description
sid	SID for the file (vnode)
task_sid	SID of the creating process
sclass	security class of this file
avcr	AVC entry reference

5.1.4 File Handle Labels

While it was not part of the original design, SELinux's use of file descriptor labels provided motivation for the TrustedBSD MAC framework to provide them as well. This recent addition provides basic support for labeling `struct file` objects within the kernel and allows for access control checks. The `file_security_struct` contains only a single field holding the SID for the object.

```
struct file_security_struct {
        security_id_t sid;
};
```

5.1.5 Network and System V IPC Labels

The SEBSD module does not yet provide labels for network objects, and the TrustedBSD MAC framework does not currently provide labeling or access control entry points for most of the System V IPC subsystem. A prototype implementing access control entry points for the System V IPC subsystem is nearly complete, and will be integrated into FreeBSD after sufficient testing.

5.2 Label Life-cycle

The TrustedBSD MAC framework manages all labels with a three state model that closely matches the life-cycle of most kernel objects; labels are initialized, created or associated, and destroyed. While all kernel objects with MAC labels have identical life-cycles, they will differ in the association and creation phases, since these often require object-specific context. For instance, the association of labels with vnodes will often be determined by the capabilities of the underlying file system, whether it is read-only or whether it supports extended attributes.

Label initialization occurs when the data structure for a kernel object is first initialized. At initialization time, SEBSD dynamically allocates storage space for per-object labels and attaches them in it's reserved label slot.

At label creation or association, a label is bound to a specific kernel object and some label fields may be completed based on context specific to the object. Label creation takes place when the module creates a new label value for a new kernel object. This is different from association, which occurs when a previously labeled object, such as a file with a persistent label, has a label associated with it. In the case of file objects, the association occurs when a persistent object is read in from disk; at this time the persistent label may also be retrieved and associated with the kernel object.

Label destruction occurs when the kernel object is no longer needed by the kernel service; at this time SEBSD frees any allocated storage for the labels.

The following table lists the various label life-cycle entry points used by SEBSD:

Entry Point	Description
sebsd_init_cred_label sebsd_init_mount_label sebsd_init_mount_fs_label sebsd_init_vnode_label sebsd_init_file_label	Initialize the label for a newly instantiated kernel object; memory for the label is allocated
sebsd_destroy_label	Destroy the label on a kernel object and free any associated memory; this common function is used for all entry points in the MAC framework that destroy object labels
sebsd_create_cred	Set the label of a newly created process credential from the parent label
sebsd_create_pipe	Set the label of a newly created pipe, using the SID from the credential of the creating process
sebsd_create_proc0	Create the process label for process 0, the parent of all kernel processes; the SID is initialized to SECINITSID_KERNEL
sebsd_create_proc1	Create the process label for process 1, the parent of all user processes (init); the SID is initialized to SECINITSID_INIT
sebsd_create_mount	Fill out the label on the mount point being created
sebsd_create_root_mount	Initialize the SEBSD security server after the root partition has been mounted; policy is located on root partition
sebsd_create_file	Set the SID of this label to the SID of the process creating the file handle
sebsd_create_devfs_device sebsd_create_devfs_directory sebsd_create_devfs_symlink	Complete the file label for the devfs_dirent being created; these entry points are called when the device file system is mounted, regenerated, or a new device is made available; calls security_genfs_sid to generate the SID for the new label

5.3 Internalize/Externalize Operations

In order to translate between kernel object labels and user space textual representations, the TrustedBSD MAC framework provides entry point functions to internalize and externalize process and file labels. SEBSD generates a string representation of labels by composing the module name with the context string; these components are separated with a '/' character. Hence, a process label would be represented as '`sebsd/root:user_r:user_t`'. The internalize entry points converts strings in this format to a SID.

The TrustedBSD MAC framework also supports internalization and externalization of labels on network interfaces and sockets, but since SEBSD does not yet address the network layer, these are unused.

The file and process entry points are as follows:

Entry Point	Description
`sebsd_externalize_vnode_label`	Produces a text representation for the file label, also used to convert pipe labels
`sebsd_internalize_vnode_label`	Produce an internal file label based on externalized label data in text format, also used to convert pipe labels
`sebsd_externalize_cred_label`	Produces a text representation for the process credential
`sebsd_internalize_cred_label`	Produce an internal process label based on externalized label data in text format

5.4 Persistent File Labels

Within the MAC framework, mounted file systems are either marked as single-label or multi-label; this distinction is made at mount time. Single-label file systems derive the label for all files from the file system mount point. Labels on single-label file systems may not be modified. For multi-label file systems, the file system is responsible for implementing a per-file source of labels. Typically this is implemented as file system extended attributes. The FreeBSD UFS1 and UFS2 file systems support extended attributes and SEBSD uses these facilities to label objects.

The devfs file system, like the Linux implementation, maintains labels explicitly through the use of genfs; the SEBSD policy specifies the labels to use.

The following table describes each of the SEBSD entry point implementations that manage persistent labels for files.

Entry Point	Description
`sebsd_copy_vnode_label`	Copy the label information, also used to copy pipe labels
`sebsd_relabel_vnode`	Change the vnode label to a new value; only called subsequent to a successful call to `sebsd_check_vnode_relabel`
`sebsd_create_vnode_extattr`	Complete the file label for the newly created file and write out the label to the appropriate extended attribute; calls `security_transition_sid` to generate the SID for the new label
`sebsd_setlabel_vnode_extattr`	Write the file label to an extended attribute
`sebsd_associate_vnode_devfs`	This entry point is currently unused
`sebsd_associate_vnode_singlelabel`	On non-multilabel file systems, this entry point sets the file label based on the file system label
`sebsd_associate_vnode_extattr`	Attempt to retrieve the file label from the appropriate extended attribute; if an extended attribute cannot be located, fallback to using `SECINITSID_UNLABELED`

6 SEBSD Entry Point Implementation

6.1 Process Entry Points

SEBSD process entry point functions manage security fields for user credentials and perform access control for process operations. The following entry points are used to enforce process access control decisions.

Since SELinux and SEBSD only allow process labels to change at program execution time, the MAC Framework relabel entry point is not used. Likewise, the relabel access control entry point, `sebsd_check_cred_relabel` is configured

to always deny requests.

The changing of process labels at execution time has proven to be sufficiently different from the behavior of other MAC policies, that the TrustedBSD MAC framework needed to be modified to support it.

This execution time label modification also tends to be incompatible with large systems, such as KDE, which uses a custom pre-binding mechanism that may not directly invoke execve(). Likewise, relying on exec-only relabeling may cause issues when Flask is ported to the Darwin kernel due to behavior of some of the Mac OS X window components; it may need to support on-demand changing of the label. When policies do permit process relabeling, the MAC Framework does offer additional protections to prevent attacks against process memory and capabilities following the subject label change.

The following table describes the permissions enforced by process entry points.

Hook	Source	Target	Permission
`sebsd_check_cred_relabel`	N/A	N/A	always deny
`sebsd_check_proc_debug`	credential	process	ptrace
`sebsd_check_proc_sched`	credential	process	setsched
`sebsd_check_proc_signal`	credential	process	signal, sigstop, sigkill, or sigchld

6.2 Mount Entry Points

The following operation enforces access control checks when performing a stat on mount structures.

Hook	Source	Target	Permission
`sebsd_check_mount_stat`	credential	filesystem	getattr

6.3 File Entry Points

The SEBSD file entry points perform access on files. The FreeBSD kernel representation of files and directories are stored in vnodes; this is the structure that contains the file label. The TrustedBSD MAC framework adds entry points to the VFS to control vnode operations, and thus control file access.

File labels are protected by the vnode meta-data lock, which must be held for

these checks. This locking requirement led to the breakup of the relabel operation into two separate checks, from and to. It is not possible to hold all vnode locks necessary to perform all access control checks in a single entry point function.

The following table describes the permissions checked for file operations.

Hook	Source	Target	Permission
sebsd_check_vnode_access sebsd_check_vnode_open	credential	file	read, write, search append, or execute
sebsd_check_vnode_chdir sebsd_check_vnode_chroot	credential	directory	search
sebsd_check_vnode_create	credential	directory file filesystem	add_name, search create associate
sebsd_check_vnode_delete	credential	directory file	search, remove_name, rmdir or unlink
sebsd_check_vnode_deleteacl sebsd_check_vnode_setacl sebsd_check_vnode_setextattr sebsd_check_vnode_setflags sebsd_check_vnode_setmode sebsd_check_vnode_setowner sebsd_check_vnode_setutimes	credential	file	setattr
sebsd_check_vnode_getacl sebsd_check_vnode_getextattr sebsd_check_vnode_stat	credential	file	getattr
sebsd_check_vnode_exec	credential	file process file	execute_no_trans transition entrypoint
sebsd_check_vnode_link	credential	dir file	search, add_name link
sebsd_check_vnode_lookup	credential	dir	search
sebsd_check_vnode_mmap	credential	file	read, write, or execute
sebsd_check_vnode_poll	credential	file	poll
sebsd_check_vnode_read sebsd_check_vnode_readlink	credential	file	read
sebsd_check_vnode_readdir	credential	dir	read
sebsd_check_vnode_relabel	credential	file filesystem	relabelfrom, relabelto associate
sebsd_check_vnode_rename_from	credential	dir file	search, remove_name rename
sebsd_check_vnode_rename_to	credential	dir file	add_name, search, remove_name rmdir or unlink
sebsd_check_vnode_write	credential	vnode	write

The SEBSD module does not currently implement the sebsd_check_vnode_revoke.

6.4 Pipe Entry Points

The SEBSD pipe entry points perform access control for interprocess communication pipes. The permissions checked for pipe objects are similar to those checked for file objects. These permissions are listed in the following table:

Hook	Source	Target	Permission
`sebsd_check_pipe_ioctl`	credential	fifo_file	ioctl
`sebsd_check_pipe_poll`	credential	fifo_file	poll
`sebsd_check_pipe_read`	credential	fifo_file	read
`sebsd_check_pipe_relabel`	credential	fifo_file filesystem	relabelfrom, relabelto associate
`sebsd_check_pipe_stat`	credential	fifo_file	getattr
`sebsd_check_pipe_write`	credential	fifo_filey	write

6.5 File Handle Entry Points

The SEBSD file handle entry points perform access control for file descriptor operations. Each `struct file` structure contains state such as the file offset and file flags for open files. Since file descriptors may be shared amongst processes with different security attributes, access to them must be controlled.

The following table describes the permissions enforced by file entry points.

Hook	Source	Target	Permission
`sebsd_check_file_create`	credential	fd	create

`sebsd_check_file_get_flags`
`sebsd_check_file_get_ofileflags`
`sebsd_check_file_get_offset`
`sebsd_check_file_change_flags`
`sebsd_check_file_change_ofileflags`
`sebsd_check_file_change_offset`
	credential	fd	use

6.6 System V IPC

The TrustedBSD MAC framework does not currently support labeling of System V IPC objects or provide access control entry points. When this kernel subsystem is fully supported by the TrustedBSD MAC framework, SEBSD will be updated.

6.7 Network Support

While the TrustedBSD MAC framework provides labeling and access control for network devices, sockets, and messages, the SEBSD module does not yet implement any of these entry point functions.

6.8 Miscellaneous Module and System Entry Points

These entry point functions are called to register and de-register the the SEBSD module. The following table lists these entry points, along with the generic system call entry point.

Entry Point	Description
`sebsd_init`	This entry point is currently unused; the SEBSD module initializes itself when the root file system is mounted
`sebsd_destroy`	This entry point is currently unused; when the SEBSD module registers with the MAC framework, configuration parameters specify that the framework should not permit the SEBSD module from being unloaded
`sebsd_syscall`	This entry point provides a policy-multiplexed system call so that SEBSD may provide additional services to user processes without registering specific system calls

The following table lists the permissions enforced by miscellaneous system and module access control entry points.

Hook	Source	Target	Permission

`sebsd_check_kld_load`
`sebsd_check_kld_unload`
`sebsd_check_kld_stat`
	credential	capability	sys_module

`sebsd_check_sysarch_ioperm`	credential	capability	sys_rawio
`sebsd_check_system_acct`	credential	capability	sys_pacct
`sebsd_check_system_reboot`	credential	capability	sys_boot
`sebsd_check_system_settime`	credential	capability	sys_time
`sebsd_check_system_swapon` `sebsd_check_system_swapoff`	credential	file	swapon

Two system entry points, sebsd_check_system_sysctl and sebsd_check_system_nfsd are not yet implemented; they control complex operations and require further research.

6.9 Entry Point Comparison

While Linux and FreeBSD both provide a similar user operating environment, the kernel services are rather different, and the kernel implementations are dramatically different. As a result, there is no convenient way to compare the LSM and TrustedBSD MAC frameworks at the entry point (hook) level. Both frameworks support labeling operations and access control checks on file and process objects, but the organization of the individual kernel services (process and VFS) caused the two security frameworks to diverge. However, since the same basic operations are being performed, the same set of permissions was applicable to both SEBSD and SELinux. Once the SEBSD prototype is complete (including network and System V IPC support), a full analysis can be done to verify that the permissions being enforced by SEBSD are comparable to those enforced by SELinux.

7 Future Development

The SEBSD module is still under development; labels are not maintained on all kernel objects supported by the TrustedBSD MAC framework, and not all access control entry points have been implemented. However, the current state is sufficient to prove that the Flask architecture is sound and that it translates well to the FreeBSD platform. Going forward, SEBSD will provide labeling support and access control checks for network objects, pipe objects, and System V IPC. In addition, the following list summarizes the items that are currently scheduled for inclusion in the next major release of SEBSD:

- Provide more complete path information in avc_audit messages. Currently the FreeBSD VFS does not provide complete path information, only file system and file identifiers are reported. There are several approaches that may be taken to improve path identification.
- Re-implement policy loading mechanisms. Taking the lead from the SELinux project, SEBSD will likely implement a user space policy loading mechanism.

- Synchronize the AVC and Security Server. Since SEBSD originally branched from the SELinux development tree, the SELinux project has made many changes and improvements; SEBSD needs to make certain all important changes made to the security server and the AVC are re-integrated.
- Synchronize user space APIs. Recent user space API changes in SELinux need to be examined in greater detail, and an attempt will be made to bring the two APIs as close together as possible.
- File system/mount point hooks are incomplete. There are some remaining technical issues with the ways the TrustedBSD MAC framework handles mount points and mount point access control checks.

8 Recent Linux Changes

Because SELinux and the LSM framework are both still works in progress, the code base originally adapted for SEBSD has diverged from the current SELinux development branch. Recent changes to SELinux include:

- All API calls were changed to pass contexts rather than SIDs. As this was already the interface that SEBSD chose, there is little impact due to this change – rather it brings the two user space APIs closer together. By having the two APIs align, it will allow easier porting of security-aware applications and Unix tools.
- Extended system calls for passing in a security context at program execution time and object creation time were replaced with a sequence of systems calls that first set a security context followed by the operation. At this point the MAC framework (and SEBSD) only makes use of the `execve_secure` extended system call; the MAC framework has added this system call. Since the TrustedBSD MAC framework does not yet permit atomic object creation and labeling system calls, the new SELinux interface will be examined in greater detail to determine whether this approach will work well on FreeBSD.
- The extended stat system calls were replaced with separate, orthogonal system calls that only obtain the security context. This is already the approach taken by the TrustedBSD MAC framework.

- All calls that return contexts or context arrays now provide automatic allocation of context buffers and context array buffers of the proper size. This simplifies the interface, as user applications no longer must guess an initial size and retry with a larger buffer upon failure. SEBSD implements many of these interfaces as sysctls, so it may not be appropriate to allocate memory on behalf of the caller – this issue will be examined in further detail.

- The policy loading API takes a `(data, size)` pair rather than a Unix path name. The SELinux module no longer directly opens files from within the Linux kernel. With the new interface, a user space application will open the file, mmap it, and call `security_load_policy` with a pointer to the policy data. Since it is preferable to avoid file access from within the FreeBSD kernel, it is likely that SEBSD will take this approach as well. It is possible that the TrustedBSD MAC framework will need to be modified to support this behavior.

For the most part, these changes will bring the user space APIs for SEBSD and SELinux closer together. The other changes to SELinux are likely to be adopted by SEBSD, and with the use of a user space security library, remaining API differences may be abstracted, allowing user applications to remain largely compatible.

9 Getting the Software

The TrustedBSD MAC Framework, as well as a number of sample policy modules, are present in the FreeBSD 5.0 distribution. This software may be downloaded from:

`http://www.FreeBSD.org/`

The MAC Framework is distributed under a two-clause Berkeley-style open source license, permitting unlimited non-profit or for-profit reuse in both open source and closed source products. Additional information on the TrustedBSD Project and SEBSD may be found at:

`http://www.TrustedBSD.org/`

Much of the LSM Framework is currently included in the current Linux development kernel releases. This software may be downloaded from:

`http://www.kernel.org/`

Additional information about the LSM Framework is available at:

`http://lsm.immunix.org/`

The SELinux software and documentation is available at:

`http://www.nsa.gov/selinux/`

References

[1] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *8th USENIX Security Symposium*. Washington, D.C., USA: USENIX, Aug. 1999, pp. 123–139.

[2] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," U.S. National Security Agency, Tech. Rep., Oct. 2000.

[3] "FreeBSD home page," FreeBSD Project, http://www.FreeBSD.org/.

[4] TrustedBSD Project, "TrustedBSD home page," http://www.TrustedBSD.org/.

[5] R. Watson, "Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD," in *BSD Conference*, Monterey, CA, USA, October 2000.

[6] ——, "TrustedBSD: Adding Trusted Operating System Features to FreeBSD," in *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[7] R. Watson, B. Feldman, A. Migus, and C. Vance, "Design and Implementation of the TrustedBSD MAC Framework," in *DISCEX III*, Washington, DC, USA, April 2003.

[8] R. Watson, W. Morrison, C. Vance, and B. Feldman, "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD," in *Usenix Annual Technical Conference*, San Antonio, TX, USA, June 2003.

[9] "Linux Security Modules home page," LSM Project, http://lsm.immunix.org/.

[10] P. A. Loscocco and S. D. Smalley, “Integrating Flexible Support for Security Policies into the Linux Operating System,” in *Proceedings of the USENIX Annual Technical Conference*, June 2001.