Design and Implementation of the TrustedBSD MAC Framework

Robert Watson, Brian Feldman, Adam Migus, Chris Vance
Network Associates Laboratories *
15204 Omega Drive, Suit 300
Rockville, MD 20850
rwatson@nai.com, bfeldman@nai.com,
amigus@nai.com, cvance@nai.com

Abstract

Developing access control extensions for operating systems is an expensive and time-consuming task. Mechanisms available for access control extension lag behind industry standard extension solutions for file systems, process schedulers, and device drivers, and suffer from a number of serious flaws in modern multi-processor, multi-threaded kernels. In this paper, we explore the limitations of current technologies for security extension. We describe the TrustedBSD MAC Framework, a flexible and modular environment for operating system access control extensions on the open source FreeBSD platform. The TrustedBSD MAC Framework permits extensions to be introduced at compile-time, boot-time, or at run-time, and provides a number of services to support dynamically introduced policies, including policy-agnostic object labeling services and application interfaces. We discuss the design and implementation of the framework, as well as the an implementation of a fixed-label Biba integrity policy based on the framework.

1. Introduction

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 of the product as the target operating system evolves. A variety of approaches for security extension exist, but all have substantial problems, ranging from specific concerns over technical correctness to high maintenance costs. The high cost and complexity of such extensions has limited the transfer of new access control technology from the security research community to the broader COTS product line, resulting in substantial and unnecessary vulnerability to common attacks. In the limited scenarios where access control technology has been successfully transferred, the product cost has been high, deployment has been difficult, and the technologies have not been widely adopted.

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. The TrustedBSD 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.

In this paper, we explore the difficulties with current access control policy implementations in COTS operating systems, the design and implementation of the TrustedBSD MAC Framework, and consider the implementations of a common mandatory access control policy. We also consider past and related work, future research directions, and mainstream operating system integration issues.

2. Background

There is a long history of operating system security research and development, in which existing operating systems are modified to support enhanced security services, and new operating systems are created with the intent of

*This work was supported in part by DARPA/SPAWAR contract N66001-01-C-8035. First published in the Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX III), Washington, DC, 2003. Proceedings published by IEEE.

offering more secure operation. In operating system security research, two focuses have prevailed: first, functionality is added to support improved security in the system through better management of privilege and improved approximation of least privilege; second, assurance of secure operation is improved through architectural enhancement and improved design and implementation processes. In the area of functionality, a variety of access control approaches have been explored, attempting to improve protection of data integrity [1], and confidentiality [2]. Likewise, extensive work has been performed in the area of evaluation for the purposes of improved assurance [3] [4] [5] [6]. However, the high cost of development, deployment, maintenance, and real-world use has limited the wide-spread use of these advanced security technologies to a small set of costly commercial and research trusted systems.

Additional research has explored how to reduce the cost associated with access control, including exploring policies with lower administrative overhead [7], avoiding modifications to the base system [8], and improving the expressiveness of policy languages [9] [10]. Research has also considered frameworks for access control instrumentation separate from the policy it enforces, including the GFAC [11], RSBAC [12], and the FLASK Architecture [13].

FreeBSD is a widely deployed production open source operating system [14]. It is used extensively in network service environments, and as a basis for many high end embedded network devices (routers, firewalls), and storage devices (storage appliances, SANs). FreeBSD development has included security features based on the scenarios in which it is deployed, but the development effort has not specifically focused on access control extension. Its wide deployment makes it an excellent target for technology transfer activities. The TrustedBSD Project is developing security extensions for FreeBSD, and integrating those changes back into the base distribution [15] [16].

3. Models for Kernel Security Extensions

Many operating system security extensions rely on modifications to the kernel to operate, preventing mandatory protections from being bypassed: all IPC and access to system resources is mediated by the kernel. Security extensions often rely on application modifications to arrange for appropriate security labeling of system objects; however, the labels themselves must be maintained by the kernel and access control checks must be performed by the kernel.

A variety of approaches have been explored in extending security services in the operating system kernel. Selection of an approach has depended on a variety of factors, including access to the operating system kernel source, the ability to make changes to that source in the vendor source tree, and the extension mechanisms supported by the kernel. A number of security extension methods rely on the availability of loadable kernel modules, the ability to dynamically link new object code to the kernel. As loadable kernel modules are supported by all major UNIX operating systems, access to this feature is a safe assumption for most security vendors. Loadable modules introduce their own security risks, as the ability to make run-time changes to kernel functionality might lead to the ability to introduce services not anticipated (and hence protected) by the security vendor, as well as the ability to bypass security protections through direct modification of control flow or data structures in the kernel.

In this section, we explore common extension strategies adopted by security extension vendors.

3.1. Direct Modification

Direct modification of existing operating system source code is the path most often taken by vendors producing a "trusted" system. It is also one of the most effective solutions for changing kernel access control policies. This strategy either bases the work on a snapshot of the operating system release, or integrates the changes into the main-line development tree. Security researchers and commercial security product developers are able to understand and modify the operating system at a fine level, as well as make changes to any part of the system that requires it.

This approach is often taken with commercial trusted systems: select a particular release of a system and apply local extensions. This has been performed on many operating systems by system vendors and third parties on commercial operating systems, as well as in research environments [17].

3.2. System Call Interposition

System call interposition modifies the kernel's system call table using a loadable kernel module, inserting new security protections between the application and kernel service. With approach, researchers avoid changing existing source code, instead introducing new security semantics by limiting access to kernel services using wrappers. Modules maintain authorization structures in parallel to those maintained by the base kernel services: prior to letting process requests reach the kernel itself, they perform their own security checks, and can limit or redirect requests in the system call wrapper. On return from the system call, wrappers may also enforce post-conditions and log activities.

This strategy has been applied both with and without source code, and can be used to introduce new security restrictions with a vendor-provided distribution. Interposition has been demonstrated for both specific policies [7], and as a more general framework for security modification [8].

3.3. Stacked File Systems

File systems store persistent data for both the operating system and applications, and as a result are common targets for security research. Security research is just one potential target of file system research requiring extensibility: reliability, namespace transformation and data transformation have all driven the development of stackable file systems. With this model, new services are "layered" over an existing file system by by wrapping operations on file system objects. In a manner similar to system call interposition, stacking permits run-time behavioral modification of a file system unanticipated by the file system author.

A variety of security extensions have been represented using stacked file systems: filtering operations permit access control, transform of credentials, etc. Namespace transforms can limit access to objects, or securely present objects. Data transforms may also be used to to provide cryptographic protections, presenting secure access to objects, or limiting access to compromised objects.

4. Limitations to Existing Extension Models

These models all have substantial limitations: some are inherent to extending complex systems, but others are properties of the extension mechanism. All assume either a lack of interest in the security extension from the perspective of the original OS vendor, or a commitment only to a narrow set of extensions. In this section, we describe limitations to these approaches that make them difficult or inappropriate to use in modern UNIX operating systems.

4.1. Required Access to Kernel Source

You must have access to the source code in order to rely on many methods of introducing new kernel access control policies. Unlike some types of operating system extension (device drivers, schedulers, file systems, network stacks), there is no well-defined API/ABI for extensions with the scope of access control. The source access requirement limits the ability to perform research, raises costs of development, and requires increased vendor involvement. It also presents limits in terms of long-term maintenance: occasional dispersions of source access are not sufficient to track architectural changes in the operating system, nor promote rapid availability of new revisions of extensions as new versions of the operating system become available.

These factors combine to make it almost impossible for non-vendors to produce timely security extensions if the vendor is not intimately involved in the process. This discourages third parties from developing security extensions that might compete with the vendor's own extensions.

4.2. Tracking Vendor System Development

Operating systems are inherently moving targets, even once a version is released. A steady stream of patches, service packs, and hot fixes to problems (especially security problems) mean that security extensions rapidly fall out of sync with their target. This is especially troublesome for extension mechanisms, such as system call interposition, which rely on a complete (and static) characterization of the system's ABI in order to operate properly.

Between official releases, active operating system development branches also move quickly; fundamental assumptions regarding system behavior change frequently. The impact of these changes on security extensions is often poorly documented, but may have a significant effect on how security extensions are integrated into the system, even if the semantics offered to applications change little. Changes may, for example, violate assumptions regarding information flow and object locking. Other than in the area of specifically published APIs and structures (such as for file systems or device drivers), security extension authors can rely on little consistency between revisions. In addition, if a security extension relies on direct change to the operating system source code, there may be literal source code conflicts in changes to code modified by both the operating system vendor and security extension vendor.

Because of the significant change between releases, or even between service packs, any formal evaluations of the operating system product with regards to a particular security extension faces substantial assurance challenges. Security vendors must make a complete argument for assurance not only concerning their own product, but also regarding the operating system vendor's product, requiring high levels of additional investment for only incremental operating system improvements. The burden lies entirely with the security extension vendor to determine that the extension will operate correctly in the new environment.

4.3. Races with Threaded Applications

Wrapping techniques, such as system call wrapping and file system stacking, are often vulnerable to attacks involving user parallelism via shared memory or threading. For example: many security policies based on system call interposition rely on atomically checking the user process arguments with the actual service performed by the system call. However, most UNIX kernel implementations pull in UNIX pathnames "on demand" from user address space in response to conditional behavior in the kernel. For a system call wrapper to make a security decision based on the system call arguments, it must independently pull in the arguments from userspace. In multi-threaded or shared memory environments, this can introduce a race, as the user application (perhaps executing on another CPU) can change the argument checked by the system call wrapper from the argument used by the actual service implementation.

An alternative employed by some system call interposition policies is to copy system call arguments and bypass the second copy performed by the kernel, re-implementing much of the kernel service. This relies on access to the source code for the original service, as well as generating duplicate code that must be maintained. Other interposition-based modules have attempted to address this problem through post-conditions, checking that the security constraints of the application have not been violated during the actual system call implementation. This cannot defend against exploitation when system calls cause side effects that cannot be backed out or (in many cases) even monitored by the system call wrapper. For example, it is not possible for a system call wrapper to back out a network communications operation on after-the-fact discovery that the application has exploited a race condition. Similar constraints also apply to implementations of audit employing system call wrappers, where the data logged may be manipulated separately from the data used to perform the access.

4.4. Lock Order and Races in Threaded Kernels

Wrapping techniques, such as interposition and file system stacking, introduce fundamental problems in environments supporting kernel parallelism: since the base system is not modified, wrappers must ensure that appropriate synchronization primitives are used to prevent time-of-check, time-of-use races within the kernel itself. In practice, this can require substantial duplication of work between the wrapper and the base component, as well as potential lock order violations and lock recursion. For example, security extensions may rely on labels on files to provide protection for those files. To access the labels, the system call wrapper must perform a series of namespace lookups to traverse the file hierarchy to find the target of the operation. Once the check is performed, the wrapper must release all locks on the file and namespace or risk violating the kernel's lock order when the service implementation attempts to perform the lookup operation. As locks are released, the namespace and protections on objects may change, resulting in a race condition between check and use. Similar races exist for all objects supporting fine-grained locking in the kernel: locks released on target processes in signal operations will permit the label on those processes to change before the kernel performs its own lookup, locking, and protection.

4.5. Interactions Between Security Extensions

Frequently, systems are deployed with several extension security policies. Typical trusted UNIX systems are shipped with the basic UNIX Discretionary Access Control policy (DAC), in addition to one or more MAC policies. For example, PitBull and Trusted IRIX both ship with Biba-derived integrity policies for TCB protection, as well as Multi-Level Security to protect the confidentiality of user data. Likewise, hardening packages are often applied in concert on deployed systems–vendor-provided security extensions may be combined with local extensions. Firewall vendors frequently run FreeBSD systems with their own local security policy variations based on their specific requirements, but rely on vendor security policies for base-line protection.

Security extensions may conflict in a variety of ways when deployed in parallel:

- Literal conflicts in source code locations: security extensions tend to modify the same system components in the same places, resulting in a conflict: both sets of modifications cannot be simultaneously applied.
- Security extensions may also have functional conflicts: models can conflict and result in an unusable or insecure system. Of particular concern are security extension that introduce new services or administrative interfaces that are not managed by other security extensions limiting access to those interfaces.
- Security extensions may interact poorly: some applications fail "closed" very badly due to lack of adaptation. Interactions between models may generate new failures that cause fail-open semantics [18].

Safe composition of extensions is particularly important in environments where extensions may be used to respond to new threats not anticipated at design time. For example, new security limits might be imposed to limit exploitation of a newly discovered application vulnerability.

4.6. Cost of Implementation

One of the most important concerns associated with all of these extension approaches is their high cost:

- High level of complexity, increasing the implementation cost, and reducing the chances of correctness.
- Different entities often implement the same infrastructure extensions performing redundant work, minimizing software reuse, and resulting in added cost.
- The "moving target" nature of the target operating system leads to a high maintenance cost, often requiring complete re-engineering of successive versions.

These high costs discourage security extension development by third party vendors and researchers, resulting in poor adoption of new (or even old) security technology.

5. Kernel Framework Approach

The TrustedBSD Mandatory Access Control (MAC) Framework is designed to address these problems in kernel security policy augmentation. As FreeBSD is open source, and the FreeBSD Project is willing to accept system extension services, we are able to adopt an approach that assumes that the operating system vendor understands the need for reliable and extensive augmentation for the purposes of security. Our primary design concerns were:

Support high levels of correctness in concurrent environments. FreeBSD has a multi-threaded SMP kernel, offering a scheduler activation-derived threading to user processes. Take advantage of the ability to modify the kernel to provide access control check and label maintenance opportunities tightly integrated with the existing locking environment, avoiding races inherent to many extension approaches.

Do not commit to a particular access control policy; rather, provide a framework that can support many common models. One need only inspect the broad array of continuing access control research and the variety of access control products to conclude that there is no wide consensus on a "one true policy" or even "one true policy language". This permits local determination of tradeoffs between security, performance, and usability during deployment.

Permit implementors of policies to make trade-offs between security and performance; provide access to traditionally costly security approaches without forcing these costs onto less demanding policies.

Permit independently developed policies to be inserted in parallel, and provide a predictable (deterministic) composition of the services. This approach reflects the reality that most commercial trusted systems include at least two different mandatory access control policies.

Augment existing kernel services to be aware of the MAC Framework, but attempt to entirely encapsulate policy-specific concerns in policy modules. While the MAC framework is derived from the requirements of specific policies, avoid leaking of policy-specific data representation or access control approaches outside of the policy modules.

Attempt to capture common policy-agnostic elements of frequently used access control models and provide access to those services in the MAC framework, avoiding frequent re-implementation for each security extension. This reflects the reality that many access control models rely on common services, such as labeling, which are complex and time-consuming to implement and integrate, presenting the opportunity for centralizing the service to reduce costs.

Provide strong support for security extensions shipped with the base operating system by the vendor, yet permit third parties to ship security extensions that drop in without the need for complicated local modifications.

6. Kernel Framework Design

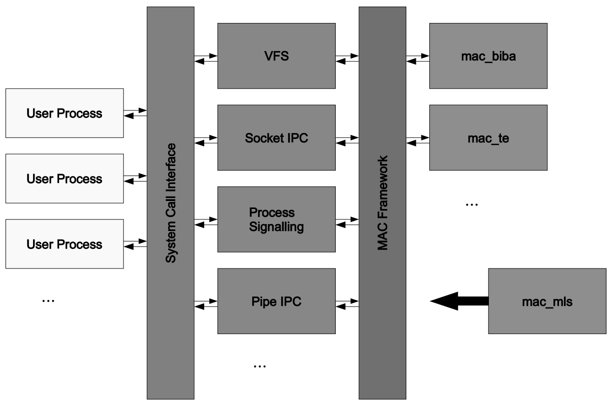

Figure 1. High-level view of the Framework

We adopt a modular approach that permits the extension of both the kernel and user applications to support new security models. This layered approach distinguishes user applications, kernel services, the MAC Framework, and a set of modules that interface with the framework.

Kernel: implement a modular framework provides common dependencies for policies (labeling), ability to augment "important" security decisions, composition of multiple modules in a predictable manner.

User: permit policy-independent security-aware applications, not just policy-aware applications, to interact with the system. Export information from the framework to permit policy monitoring and common administration elements.

6.1. MAC Interface to Kernel Services

The MAC Framework presents a set of entry points to selected kernel services, permitting the services to provide event notification to the MAC framework, and providing the ability for the MAC Framework to maintain a security label within kernel objects maintained by the kernel services.

6.2. MAC Interface to Security Policies

The MAC Framework provides several interfaces to security policy implementations, including interfaces for policy management, label storage, process label management, object life cycle, access control, and system life cycle. Extensions implement arbitrary subsets of the available interfaces, allowing implementors to select the events and services that are relevant to a particular policy.

6.3. MAC Interface to User Processes

The MAC Framework provides user APIs that reflect operations commonly exported by policies. These APIs include interfaces to manage policies loaded and enabled in the kernel, and a flexible labeling API for retrieving and setting security properties of relevant kernel objects.

7. Kernel Framework Implementation

7.1. Components of the MAC Framework

The TrustedBSD MAC Framework is split into a number of logical components:

- MAC Framework Interfaces for Kernel Services. The interface used by FreeBSD kernel services to communicate with the MAC Framework is defined in sys/mac.h. This includes the APIs for all entry points from the kernel services. In addition, sys/_label.h defines `struct label`, a data structure used to store policy-agnostic label data in kernel objects. This structure is embedded into many kernel service structures.

- Framework Kernel Service Entry Points. Modifications have been made to kernel services to invoke MAC Framework entry points. These modifications affect object initialization, association/creation, and destruction, as well as in common paths requiring access control at high levels in the kernel. With layered services, it is often necessary to defer access control decisions until enough information is available.

- Framework Implementation. Entry point implementations, label primitives, policy registration, and user/kernel APIs are centralized in kern_mac.c.

- Framework Interface for Policies. Interfaces common to the framework and policies and defined in sys/mac_policy.h. Definitions include entry point and registration interfaces, as well as common access methods for MAC Framework services.

- Policy Implementations. Each policy is represented by one kernel module, discouraging inter-dependency. Typical policies are implemented in a single C file, but complex policies are implemented over many files.

- Interfaces to User Processes. Interfaces for user processes are defined in sys/mac.h, implemented in `libc`, and may be dynamically linked into any applications.

7.2. Framework Startup

The framework is initialized early in the boot process, shortly after the kernel memory allocator, console, and locking primitives, but before high level device probing and any kernel or user processes have started. The MAC Framework initializes itself, registers policies, and supports other kernel services that provide relevant kernel objects.

Initialization prepares various administrative structures, including the policy registration structures and locking primitives. Following initialization, early registration of policies is permitted: this applies to any policy linked into the kernel itself, or loaded by the boot loader prior boot. Early policies have the opportunity to ubiquitously label devices and kernel data structures as they are allocated or probed. After early policy registration, a variable is set to indicate that any policies registered after that point will not be able to ubiquitously label kernel objects.

Figure 2. Policy life cycle

7.3. Policy Registration

Policies register with the MAC Framework to receive events, reserve label space, and access MAC Framework services. Modules are distinguished from policies: kernel modules may contain a number of code objects, and may encapsulate multiple policies. Policies register with the framework once their modules have loaded. Each policy declares a number of properties, including policy name, vendor, version, label requirements, load-time flags, and entry point definitions. The load-time flags indicate whether or not the policy may be attached after boot, and whether it may be unloaded. Policy registrations are protected by a busy count and lock; changes to the set of registered policies requires that all in-progress entry points complete, providing consistent enforcement of policy during registration changes.

7.4. Entry Point Invocation, Composition

In almost all cases, invocation of a MAC Framework entry point from a kernel service invokes matching entry points for the set of registered policies. Policy entry points are split into three categories based on the wrapper macro used to invoke them and compose the results:

MAC_PERFORM assumes that a policy entry point has no return value, and is used to post an event to interested policies. Events may related to policy changes, label management, policy management, or kernel object life cycle events. No explicit return value composition is required.

MAC_CHECK implements a call-out to an access control entry point, or an entry point supporting detection and classification of failure modes. Unlike MAC_PERFORM, it accepts an errno return value from each policy and composes the results using error_select, a function that encodes an ordering of various failure classes.

MAC_BOOLEAN implements a call-out to a decision function, and composes the return value using an arbitrary boolean operator. This is used in a number of special case scenarios where policies augment an existing kernel service decision rather than returning an access control result.

7.5. Access Control Entry Points

Entry points are scattered throughout the kernel, including process management, file systems, IPC, and the network stack. In general, access control entry points follow a similar form, accepting the access control context (requesting credential, target object(s), and other relevant arguments), and invoking the entry point for each policy, returning 0 for success or a failure value. Results from invoking entry points for each registered policy are are composed by the framework using a simple composition policy. The composition requires all policies to accept a check for it to succeed, and selects one of the errors based on precedence if a check fails.

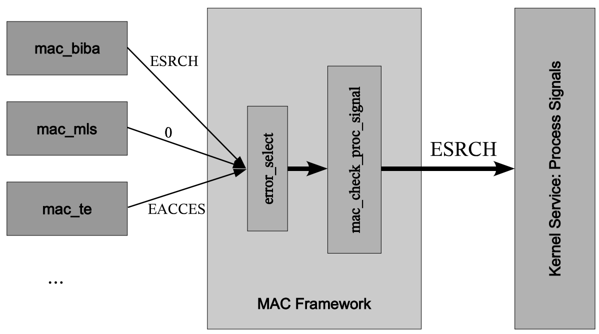

Figure 3. Policy composition

7.6. Label Management

A number of access control policies rely on security labels maintained on objects for the purpose of performing access control decisions. For policies such as MLS and Biba, labels are provided explicitly by the administrator or derived from the run-time environment. Their labels store sensitivity of integrity information about a kernel object, using this information during decision-making. For other policies, labels might consist of data types, rule sets, or a other information. The framework supports labeling for several classes of security-relevant kernel objects, including several file systems, processes, and network stack kernel elements.

Structure	Description
struct ucred	Process credential
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

Figure 4. Labeled objects

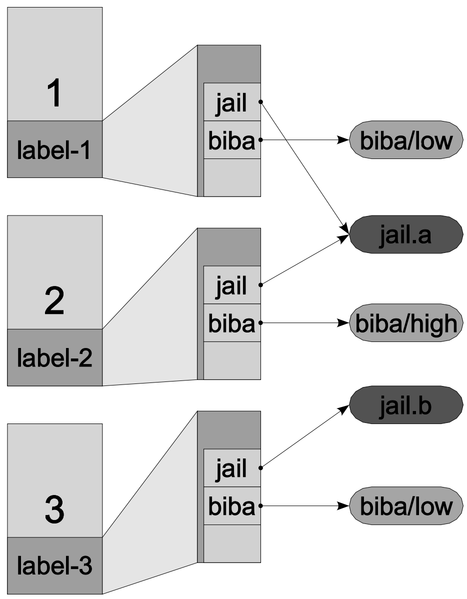

Figure 5. Object label semantics

Each structure holds one or more instances of a policy-agnostic label structure for per-policy data. This structure is intended to provide maximum semantic flexibility for policies to implement desired behavior: each policy reserving label state is allocated a slot in the label structure, and each slot holds either a void pointer and an integer. Policies may implement per-label storage, reference-counted storage, or statically allocated storage. Kernel objects are pooled by the kernel slab allocator so that expensive memory allocation costs can be amortized. Labels are initialized, allocated, and destroyed along with their object:

Label initialization occurs when the data structure for a kernel object is initialized, permitting policies to allocate and initialize memory for the object. The kernel distinguishes two types of object allocation: where the caller permits blocking, and where the caller cannot tolerate blocking; for example, blocking while holding sensitive locks is discouraged. This semantic is present for labels also: most label initialization is permitted to block, but for specific objects the caller can request a failure over blocking.

Label association or creation occurs when an initialized kernel structure is associated with an actual kernel object, such as a file or process. Association refers to the binding of a kernel object structure to an existing external or persistent object that already has labeling data present in an externalized form. Association typically occurs when a persistent object is faulted in and its label internalized; for example, an existing file on disk. Creation refers to the binding of a kernel object structure to an entirely new instance of an object. In this context, a specific subject label is available for the purposes of calculating a fresh label for the new object. Memory allocation by policies is discouraged under either circumstance, as locks are often held during object creation, making extended blocking for memory undesirable. Instead, policies should allocate all memory required during label or policy initialization. Association and creation may be permitted to fail under some circumstances: for example, an invalid label in a persistent label store, or insufficient resources to create persistent label store for a new object; when this happens, the object creation also fails.

Label destruction occurs when kernel object is released by the kernel service implementing it. The MAC Framework is given the opportunity to release storage for the label, permitting policies to free any allocated storage or references associated with that label. This might reflect the destruction of an actual object, or simply the recycling of the in-memory storage for a persistent object.

7.7. Per-Object Behavior

While all kernel objects with MAC labels have similar initialization and destruction entry points, labels are handled differently across object classes in a several ways, including creation, object-specific life cycle events, and label handling procedures (such as locking).

Association and creation events depend on the availability of context, and vary not only by object class, but also for specific instances of those objects. For example, file system vnodes are handled in a manner specific to the file system, based on whether or not the file system supports individual labels for individual vnodes.

As with all kernel object elements, MAC Framework-maintained labels must be protected against unsynchronized parallel access by means of locking primitives and access protocols. In general, the MAC Framework relies on native locking protection for objects to protect the labels on those objects. For example, vnode labels are protected by the vnode lock, whereas mbuf labels are protected by virtue of consistent serialized access by reference holders.

7.8. Credentials

Process credential structures stores identity and authorization information associated with the UNIX security model. With the introduction of the MAC Framework, a label field is introduced. As with other data fields in the credential, the label field is protected by an “immutable once shared” policy–the credential label may be changed only while one reference exists, such as immediately after creation. All credentials are derived from either process 0 (the parent of all kernel processes), or process 1 (the parent of all user processes). Processes have a primary credential pointer, protected by the process lock; individual threads may have additional thread-local credential pointers that are synchronized to the process credential pointer value on entry to the kernel. This provides credential labels with the same consistency properties as other credential data in a threaded environment, and ensures that credential used by a thread is consistent for the duration of a system call.

7.9. VFS Objects

A variety of VFS objects are supported by the MAC Framework, including the file system mountpoint, vnodes representing objects inside the file system, and devfs prototype structures which represent system devices exposed via /dev. Mounted file systems are marked either single-label or multi-label. Single-label file systems are presumed not to have a source of per-vnode labeling data, and the labels for all vnodes are derived from the file system mountpoint by the framework. Labels on single-label file systems may not be modified. For multi-label file systems, the file system is responsible for implementing a source of labels and performing association and creation operations to ensure that labels are always properly defined for file system objects.

The MAC Framework provides a centralized implementation of extended attribute based label storage that file systems supporting extended attributes (such as UFS1 and UFS2) may employ to label objects. For devfs special handling is provided by the framework to permit policies to maintain labeling data for system devices explicitly as part of their configuration. In a manner similar to association and creation entry points, the MAC Framework also relies on a per-file system implementation of `VOP_SETLABEL`, which pushes a label change to the vnode's persistent label store, and then updates the vnode label if successful.

Vnode labels are protected by the vnode reader/write lock, which is generally already held where consistency is required for enforcement. The framework relies on the compound write functionality for UFS2's extended attributes, combined with the soft updates consistency model, to ensure consistency properties of on-disk labels.

7.10. Sockets

Sockets are the basic IPC primitive associated with the BSD IPC model. They represent two different object classes: a communications endpoint, and a communications rendezvous. Each socket has two labels, an object access control label, and a cached copy of the remote endpoint (peer) label. Sockets largely derive their labels from the processes that create them. For accepted sockets associated with incoming connections, the socket object label is typically derived from the listen socket from which it was created. Peer labels are set when the first incoming datagram is received on network stream sockets, or from the remote socket endpoint on local UNIX domain sockets.

7.11. Mbufs

Mbufs represent datagrams "in flight" in the network stack. Mbufs are a data allocation structure that has been designed specifically for the operations found in network stacks (header prepend, iteration through data). Some mbufs store packet header data, includes information on the packet carried by the mbuf or mbuf chain. The TrustedBSD MAC Framework stores label data in the optional packet header, which avoids unnecessary label storage while providing labeling opportunities for all relevant packets. Mbuf labels are typically derived from their originating socket or network interface, or from another mbuf.

7.12. Network Interfaces

Network interfaces represent sources of non-local packets, as well as a routing destination for local and non-local packets. An interface might be a physical Ethernet interface, a virtual interface such as a PPP connection or VLAN, or it may represent a communications tunnel endpoint. As with sockets, a label is present in the structure to provide labeling information for packets sourced by the interface, as well as to perform access control for packets destined for the interface. Interface labels are determined when the interface is discovered, either as a result of a hardware probe, or an administrative action in the case of virtual interfaces.

7.13. Application Interfaces

The TrustedBSD MAC Framework provides a variety of policy-agnostic interfaces for applications that are label-aware or policy-aware. The basic application interfaces are available via the standard C library. Interfaces exist to retrieve and set labels on file system objects (vnodes), sockets, pipes, and network interfaces. Above the library API, labels are handled by applications as strings, which may be opaquely read from or written to files, or through direct interaction with the user. The string representation of the label reflects the flexible nature of the labeling infrastructure: policies claim ownership of elements of the label by name, internalizing and externalizing the string components for use inside the kernel. The MAC Framework itself is aware only of the element names and data strings, not the semantics associated with the strings. A typical label managed by a user application might like like `biba/low,mls/10` representing a label consisting of two elements: a low integrity biba label, and an MLS label of sensitivity "10". Applications may address all of the elements available on an object, or any subset.

8. Kernel Policy Module Design

TrustedBSD MAC policies are encapsulated in loadable kernel modules. At the option of the developer or administrator, a policy module may be linked to the kernel at build-time, loaded prior to kernel start at boot-time, or loaded at run-time. Policy modules are permitted to define a set of properties that determine which of these times are appropriate based on the nature of the policy. Some policies require ubiquitous access to all system objects; for locking reasons, this access can only be guaranteed if the module is present from inception, preventing run-time loading of the module without a reboot. Other policies may be loaded at any time. Policy modules typically follow a common structure:

They contain a description structure specifying the load-time flags, name and label information, and references to other policy structures. The structure is processed by the MAC Framework when the policy is initialized.

They contain a mapping of policy entry points to the functions that implement them. Policies may implement an arbitrary subset of the entry points; developers may provide a function for each entry point, or implement entry points with identical prototypes using the same function.

They often define a number of administrative toggles determining aspects of their behavior. Toggles might configure debugging, or be used to configure policy rules.

They typically isolate label management from the implementation of their policy, relying on a set of common access control primitives to implement most entry points.

Policy	Description
mac_biba	Hierarchal fixed-label integrity
mac_bsdextended	"File system firewall" using existing credentials/permissions
mac_ifoff	Interface silencing
mac_lomac	Hierarchal floating-label integrity
mac_mls	Multi-Level Security with compartments
mac_none	Prototype stub policy
mac_partition	Inter-process visibility policy based on process partition labels
mac_seeotheruids	Inter-process visibility policy based on existing credentials
mac_test	MAC Framework invariant tests
sebsd	Port of the SELinux/FLASK/TE

Figure 6. Table of sample policies

Access control logic is usually centrally located in the module source files to make it easier to understand and modify.

9. Kernel Policy Module Implementations

FreeBSD 5.0 includes a number of kernel modules that provide a diverse set of kernel access control extensions. Extensions are also available from third parties. For the purposes of exploring the functionality of the TrustedBSD MAC Framework, we consider mac_biba, an implementation of the widely used fixed-label Biba integrity policy.

9.1. TrustedBSD Biba Policy

The Biba integrity policy implements a strict information flow policy limiting the impact of lower integrity subjects and objects on higher integrity subjects and objects [1]. In general, high integrity subjects are allowed to write but not read lower integrity objects, and low integrity subjects are allowed to read but not write high integrity objects. Biba relies on ubiquitous labeling of all system objects, and determines access control results with a dominance operator over label pairs. The policy has been frequently deployed in trusted systems to protect the Trusted Code Base (TCB).

The TrustedBSD Biba policy module implements a hierarchal and compartmental, fixed-label mandatory integrity policy. Subject labels contain an effective label element, as well as a range of available elements. Objects in the system are labeled only with a single element. When a label is initialized by the framework, memory is allocated to store Biba elements. When objects or subjects are created, label values are inherited from the parent subject. When objects are loaded from persistent or external store, labels are assigned based on the of the storage medium. Labels on system objects, such as network interfaces, are derived from kernel tunables, but may also be managed using user tools.

This policy uses central access control logic. The dominate function compares two Biba elements with respects to their types, grades, and compartments to determine their relationship. Entry points categorize access checks regarding information flow from the subject to the object, or object to the subject, and then invoke dominance tests.

The registration flags indicate to the framework that the policy must not be loaded late (i.e., after kernel objects have been allocated), and that the policy may not be unloaded, preventing the Biba policy from being removed once it is attached. The Biba policy provides a pointer to its label slot information, requesting that the framework allocate a label slot for use by the policy, which may later be used to dereference policy-agnostic labels passed via entry points.

The TrustedBSD MAC Framework successfully isolates the details of the policy implementation from the kernel services, but also isolates the details of the kernel services from the policy implementation, reducing the risks of minor changes in one subsystem requiring gratuitous changes in the other. The MAC Framework supports the Biba policy through a generalized label management service, permitting the policy implementor to focus on the details of the policy application rather than the mechanism of instrumenting the kernel to support the policy's instrumentation requirements.

10. User Application Approach

For the purposes of the MAC Framework, applications may be sub-divided into three categories:

Applications that are not aware of any non-UNIX access control policies. From the perspective of this class of applications, no specific adaptation is required. Applications may become aware of the MAC Framework as a result of new behavior in the system (most frequently, new failures), but this is purely a property of the policies registered with the framework and currently active. As with all security extension mechanisms, policy authors must be careful to avoid unanticipated consequences of system behavior changes. For example, policy authors must consider the potential impact of causing a system call to fail, where normally it would succeed unconditionally. A number of security vulnerabilities have been present in shipped systems where application expectations for system behavior were violated as the result of a "security improvements" [18].

Applications that are aware of the MAC Framework, but unaware of specific policies. These applications are typically system applications that are either used to monitor and set object labels, or interact with user credentials. These programs treat labels in an abstract, policy-agnostic manner. The userland framework relies on a configuration file,

/etc/mac.conf, to determine administrator-defined defaults for labels to query and list on files, interfaces, and processes using the existing `ls`, `ifconfig`, and `ps` programs. In addition to these basic UNIX tools, the login and credential management libraries also handle labels at an abstract level, permitting them to set process and tty labels at login based on user login class entries.

Applications that are aware of a specific set of policies. Depending on the nature of the the application, developers may choose to use the policy-agnostic interfaces provided by the MAC Framework, or new policy-specific interfaces exported specifically by the policy. For example, applications that are aware of the semantics of MLS labels may perform labeling operations involving only MLS label elements using policy-agnostic labeling interfaces. On the other hand, the mac_bsdextended policy module exports a rule list via the kernel sysctl management interface.

11. Related Work

As described in the Background section, operating system access control research and commercial development have long histories. There has been extensive research into access control policy, the implementation of access control extensions, and the impact of access control changes on the system and applications. Work related to flexible access control extension environments includes:

Generic Software Wrappers permit run-time extension of the kernel security environment using system call interposition. The implementation includes a portable cross-platform wrapper specification language, as well as tools for distributed management [8].

Security-Enhanced Linux is an adaptation of the FLASK access control framework with MLS, RBAC, and TE policies to the Linux operating system. FLASK provides policy implementations with a high level abstraction of object labels (referenced by Security Identifiers), object methods, and an Access Vector Cache which improves performance by caching security decisions from a policy engine [9] [19]. Recent versions of SELinux have relied on LSM to attach to the Linux kernel. The TrustedBSD Project includes a task to produce a prototype port of the SELinux FLASK/TE environment to FreeBSD using the MAC Framework.

Rule-Set Based Access Control (RSBAC) provides an extensible security environment for the Linux operating system, including a variety of policy modules [12].

Linux Security Modules (LSM) is a security extension framework for the Linux operating system providing a hook framework to permit new security modules to instrument parts of the kernel. This framework provides low level access to kernel operations, as well as void pointers that may be used by a single security module to store data in kernel objects. Recent work has focused on integrating the framework into the base Linux kernel, and there has been some investigation of stacking modules to compose policies [20].

In prior TrustedBSD work, a variety of security extensions, including many of the security policies now available via the MAC Framework, were implemented via direct modification of the operating system kernel [15] [16].

12. Future Directions

The TrustedBSD MAC Framework opens up many opportunities for future research and development, relating to new services and capabilities made possible by the framework, and for expanding the scope of the framework. Additional study and development of the current system should involve performance analysis and optimization, which will play an important role in possible adoption of this technology. Current work has focused purely on access control; audit of security events could be an important future capability.

One possible research direction moves beyond the "restrictive" model taken in the MAC Framework. Currently, policies act in concert to restrict access to system services; exploring more flexible composition methods would open the door for a broader range of policy interactions. In addition, this might permit the UNIX security policy to be placed in policy modules itself, allowing existing security services to be removed and replaced. Another direction for future work involves expanding the capabilities of the framework to support live policy replacement: rather than releasing label data on the unload of a policy, permit the atomic replacement of a policy so that the capabilities of the policy can be upgraded or modified.

The MAC Framework opens the opportunity for dynamic OS policy change in response to environmental change. Currently, loading and modification of policies is administrator-driven, but the framework could support an automated response capability permitting the system to adapt to new requirements or vulnerabilities.

Finally, additional research must be performed in the area of long-term maintenance and assurance properties of flexible access control approaches. As the MAC Framework provides a set of interface guarantees relative to policy implementations, the task of assurance is now split differently between the operating system and security extension providers. We argue that the MAC Framework approach will ease the burden of in maintaining security extensions: it may also lower the burden for assurance activities.

13. Getting the Software

At the time of the writing of this paper, the TrustedBSD MAC Framework is under final development and initial deployment for the release of FreeBSD 5.0. The TrustedBSD

MAC Framework, as well as a number of sample policy modules, will be 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 may be found at:

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

14. Conclusion

The TrustedBSD MAC Framework provides a flexible mechanism for compile-time, boot-time, and run-time augmentation of the FreeBSD kernel access control policy. It supports policies in a variety of ways, including instrumentation of important kernel access control decisions, centralized labeling support, and policy composition. Through tight kernel integration, the framework permits safe operation in highly parallel and threaded kernel and user environments. By avoiding a commitment to a specific access control policy, this approach permits the operating system vendor, third party distributors, and local system maintainers to independently augment the access control environment in response to local and global security requirements. By permitting dynamic run-time changes in the security environment, systems equipped with the MAC Framework can be used as the basis for more effective future operating system security research and development.

References

[1] K. Biba, "Integrity considerations for secure computer systems," Mitre, Bedford, MA, Tech. Rep. TR-3153, Apr. 1977.

[2] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations and model," The MITRE Corp., Bedford MA, Tech. Rep. M74-244, May 1973.

[3] *Trusted Computer System Evaluation Criteria*. U. S. Department of Defense, December 1985.

[4] N. C. C. I. Board, "Common criteria version 2.1 (ISO IS 15408)," 2000.

[5] N. S. A. Information Systems Security Organization, "Controlled access protection profile version 1.d," October 1999.

[6] ——, "Labeled security protection profile version 1.b," October 1999.

[7] T. Fraser, "LOMAC: Low water-mark integrity protection for COTS environments," in *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[8] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *IEEE Symposium on Security and Privacy*, 1999, pp. 2–16.

[9] 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.

[10] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker, "A domain and type enforcement UNIX prototype," in *Computing Systems, Winter, 1996.*, vol. 9, no. 1. Berkeley, CA, USA: USENIX, Winter 1996.

[11] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson, "A generalized framework for access control: An informal description," in *Proc. 13th NIST-NCSC National Computer Security Conference*, 1990, pp. 135–143.

[12] "Rule set based access control (RSBAC) for linux," http://www.rsbac.org/.

[13] 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.

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

[15] R. Watson, "Introducing supporting infrastructure for trusted operating system support in FreeBSD," in *BSD Conference*, Monterey, CA, USA, October 2000.

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

[17] "Argus products overview: Pitbull," Argus Systems, http://www.argussystems.com/product/overview/pitbull/.

[18] "Linux capabilities vulnerability," SecurityFocus, 2000, http://www.securityfocus.com/bid/1322.

[19] 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.

[20] "Linux security modules," http://lsm.immunix.org/.