The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0

Robert Watson
Network Associates Laboratories
Rockville, MD
rwatson@nailabs.com
http://www.nailabs.com/

Wayne Morrison
Network Associates Laboratories
Rockville, MD
tewok@tislabs.com

Chris Vance
Network Associates Laboratories
Rockville, MD
cvance@nailabs.com
http://www.nailabs.com/

Brian Feldman
FreeBSD Project
Rockville, MD
green@FreeBSD.org

Abstract

We explore the requirements, design, and implementation of the TrustedBSD MAC Framework. The TrustedBSD MAC Framework, integrated into FreeBSD 5.0, provides a flexible framework for kernel access control extension, permitting extensions to be introduced more easily, and avoiding the need for direct modification of distributed kernel sources. We also consider the performance impact of the Framework on the FreeBSD 5.0 kernel in several test environments.

1 Introduction

Access control extensions have proved a fertile field for operating system security research over the past twenty years: a variety of methods have been employed to extend the system access control policy at great cost to the developers, maintainers, and users of the extended systems. Most of approaches to security extension fall short two vital areas: lack of support by the operating system vendor for various providers of security extensions on the system, and the highly redundant implementation of support infrastructure for security extension providers.

The TrustedBSD MAC Framework included in FreeBSD 5.0 provides a general facility for extending the kernel access control policy [8][17]. By providing common security infrastructure services, such as kernel object labeling, and the ability to instrument kernel access control decisions, the Framework is capable of supporting a variety of policies implemented by different vendors. This paper explores the design, implementation and performance of the MAC Framework, as well as its impact on policy design and the FreeBSD kernel architecture.

2 The Desire For Access Control Extensions

FreeBSD serves two primary markets: it is both a consumer operating system and a technology source for third party operating systems or high-end embedded products. FreeBSD is directly employed as a production server and workstation operating system on mainstream i386, Alpha, and SPARC64 hardware. In the high-end embedded market, it is used as the basis for network and storage appliance devices such as firewalls, network-attached storage, and VPN devices; it is also used as a technology source for third party operating system, including Apple's Mac OS X Darwin kernel, as well as by other operating system vendors.

In these three roles, FreeBSD is deployed in a wide variety of environments, ranging from electronic cheque processing and point of sale devices to web cluster deployment, firewall, and routing appliances. Each of these environments has different security requirements, often requiring flexibility beyond that provided for by the traditional UNIX security protections. Especially in embedded network environments, the requirements can range from simple operating system hardening to the introduction of mandatory and fine-grained security policies.

3 Access Control Extension Mechanisms

Security research and development literature is rife with approaches to achieving operating system access control extension with (and without) the help of the operating system vendor. Traditional trusted variants of commercial UNIX operating systems have been written by the vendor in response to the needs of specific consumers (such as US DoD) [12][5][9][10]. Typical practice has been to maintain a distinction between the base OS product and the trusted variant in terms of maintenance, product identification, and price. In addition, there are third party vendors who develop and market trusted operating system extensions, often in close coordination with the OS vendor[2]. Finally, there is a broad range of access control research across many operating systems and performed in many forms [7] [13] [15]—most frequently, this work is performed on open source operating systems due to ready access to operating system source code.

In order to successfully maintain a security extension product for an operating system, access to the operating system code is typically required, be it an open source system, or licensed from the closed source vendor. Product maintenance raises a number of challenges, not least the challenge of tracking the operating system vendor's primary product life cycle, which is frequently incompatible with the development cycle required for high assurance products. Many practical impediments also present themselves: security extensions have their fingers deep in the heart of the operating system, touching almost all elements of the kernel source code. Local security extensions invariably conflict with vendor-provided security patches, as well as vendor-provided feature improvements over the OS development cycle. In addition, security extensions frequently conflict with one another if deployed in parallel, leading not only to potentially inconsistent policy behavior, but also possible bypass of protections provided by one of the policies. Direct source code modification of the vendor operating system presents many challenges to security extension authors.

In the past, research has been performed on how to most easily extend operating system security policies, including into system-call interposition technologies such as LOMAC [6], Generic Software Wrappers [7], and systrace [**?**], extensible security mechanisms such as the General Framework for Access Control [1] and FLASK [15]. Many of these extension technologies, especially these using system call wrapping techniques, fall down in the face of modern UNIX operating system kernels which support true kernel and user process parallelism in SMP environments, and fine-grained threading of user processes. Preventing races inherent to system call wrapping is difficult, and most system call wrapper security technologies are susceptible to at least one of a class of related vulnerabilities. Any successful extension technology for contemporary operating systems must be designed with the notion that SMP and threading are realities, and must be well-integrated into the kernel locking mechanisms.

4 Motivations for MAC

Mandatory Access Control (MAC) describes a broad class of access control policies; in this context "mandatory" refers to the mandatory imposition of the policy on non-administrative users. Popular mandatory policies including Multi-Level Security (MLS), which enforces mandatory protections based on administrator-defined confidentiality labels, Biba integrity, which enforces system and user data integrity properties, and Type Enforcement, which permits the administrator to define subject domains, object types, and use a policy language to control accesses to objects and other system properties.

Trusted operating systems typically provide two or more mandatory system policies: almost all provide MLS for user data protection, but many also make use of the Biba policy to protect the integrity of the Trusted Code Base (TCB). The TrustedBSD Project, in seeking to provide access to trusted operating system features, provides several MAC policies for use with FreeBSD; the challenges associated with this work include introducing the services securely, and without substantially impacting the performance and reliability of FreeBSD installations not taking advantage of these new features.

5 Framework Design and Implementation

The TrustedBSD MAC Framework permits access control policy modules to be loaded into the FreeBSD kernel, providing a tightly integrated security extension vehicle. In order to address the problems identified in Section 3 there are several high-level goals for the design:

- Permit dynamic extension of the kernel access control policy.
- Isolate the logic of access control policies from the implementation of kernel services, permitting the implementations of services to be more mobile in the face of extensions, and reducing OS life cycle issues for policy developers.
- Permit multiple policies to be loaded simultaneously with some useful notion of composition.
- Reduce the redundant infrastructure implementation efforts of policy writers by providing support for common policy infrastructure requirements.
- Integrate tightly with the kernel locking and threading mechanisms to provide correctness and high performance in modern kernel designs.

5.1 High Level Design

The MAC Framework is made up of a number of kernel and user-space elements. In the kernel, existing kernel services are modified to add data structure extensions and entry points to the MAC Framework, centralized management of label storage, registration and management of security modules, a series of system calls and sysctls to permit applications to interact with labels and manage the framework, and a series of access policy modules that may be compiled into the kernel or loaded via loadable kernel modules. In user-space, several new C library interfaces provide access to centralized label configuration via `mac.conf`, and changes to the `libutil` user class and security context management code. Command line tools permit user manipulation of file and process labels, and modifications to standard administrative tools manage system labels.

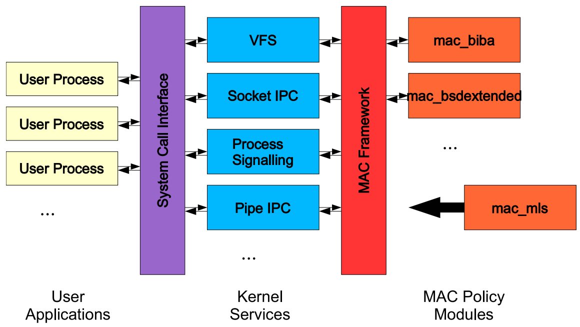

Figure 1: High Level Kernel Design

The MAC Framework addresses a number of needs in access control and extension implementation:

- Policies are encapsulated in kernel modules, which may be linked into the kernel, loaded as part of the boot process, or loaded at run-time in response to environmental requirements.
- Policies are permitted to augment kernel access control decision; sufficient locks to access important elements of check arguments, such as object references, are guaranteed to be held.
- The MAC Framework provides a policy-agnostic labeling service permitting policies to maintain additional meta-data on a variety of system objects. Well-defined locking semantics are provided for object labels, and existing locks on kernel objects typically also protect any labels in the object, permitting atomic checks of both labels and existing object properties without additional locking overhead.
- Policies may back labels into persistent extended attributes provided by UFS and UFS2, permitting labels on file system objects to be maintained while they are not in the in-memory working set.
- When multiple policies are loaded, their access control decisions are usefully composed, where the definition of "useful" is that results are well-defined, may be reasoned about, and are desirable in the context of a number of relevant policies.
- Policies may make use of a policy-agnostic label management API to export access to label data to user processes, as well as permit the management of those labels.

5.2 Kernel Services and Objects

The MAC Framework enforces policy over a variety of kernel subsystems and objects, including system configuration interfaces, processes, the file system, IPC primitives, and the network stack. In general, two classes of modifications were made to existing kernel service providers. First, services are modified to invoke MAC Framework entry points during object management, over the course of object life cycles, and when important access control events occur. Second, a number of kernel data structures representing security-relevant objects were modified to include a label structure intended to hold extensible security information.

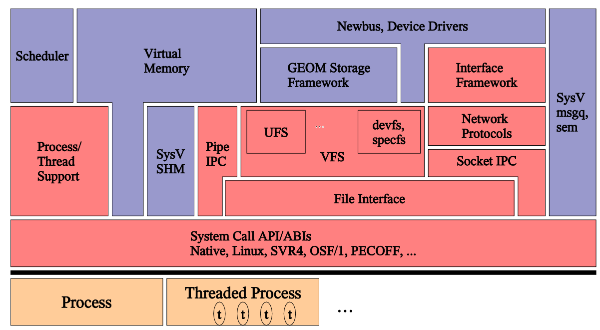

Figure 2: MAC Framework: Integration into Kernel Components

5.3 Entry Points

MAC Framework entry point invocations are conditionally compiled into kernel subsystems based on the configuration parameter `options MAC`. Several classes of entry points exist, including label management, event notification, decision functions, and access control checks. All entry points accept contextual information; typically this includes a subject process credential and a series of as objects, object label pointers, and call-specific arguments such as signal numbers or blocking disposition.

Some entry points, such as access control checks, return error values; other notification entry points are assumed always to succeed. Frequently, a set of related entry point invocations will be made around complex operations: for example, access control checks are required to create a new object in the file system namespace. Likewise, when label modifications occur, a two-phase commit is performed by the Framework to confirm that all policies will permit the relabel, and then to notify all policies to perform the actual operation.

Entry points are currently found in the cross-file system VFS code, device file system, mount/umount code, protocol-independent socket calls, pipe IPC code, BPF packet sniffing code, IP fragment reassembly, IP socket send and receive code, network interface transmission and delivery, credential and process management code (including debugging, scheduling, signaling, and monitoring interfaces), kernel environmental variable management, kernel module management, per-architecture system calls, swap space management, and a variety of administrative interfaces such as time management, NFS service, `sysctl()`, and system accounting. These entry points permit policies to augment security decisions in a variety of forms.

5.4 Labels

While some system hardening models employ existing subject and object information (UNIX credential data, file permissions, ...) a number of important mandatory policies require additional subject and object labeling. For example, the MLS confidentiality policy makes decisions based on subject and object sensitivity labels: subjects are assigned clearances, and objects are assigned classifications. When policies require additional labels, the MAC Framework supports them through a policy-agnostic labeling primitive, which permits policies to tag kernel objects with information required for policy decision-making.

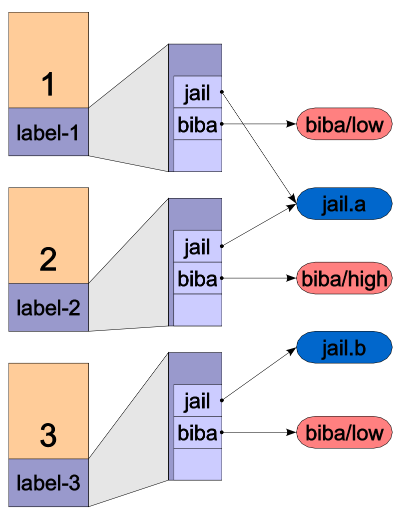

Figure 3: MAC Framework: Policy-Agnostic Label Storage

The label structure stored in kernel data structures is maintained by the MAC Framework: based on the life cycle of the data structure, the Framework provides per-object entry points for memory initialization, object allocation, and object destruction. The label structure consists an array of slots, each providing a union of a `void *` pointer and a `long`; slots are allocated to policies requiring label storage for use in a policy-specific manner. Policy writers might choose to store an integer value, allocate per-label memory, or make use of referenced structures relying on the initialization and destruction calls to maintain reference counts. Initialization calls will often be used for memory allocation, and in some cases a blocking disposition will be passed as an argument to the call indicating whether blocking allocation is permitted; in these cases the initialization call is permitted to return a memory allocation failure, which will abort the allocation of the object.

Label storage is currently provided in the following kernel objects: BPF descriptors, process credentials, devfs directory entries, network interfaces, IP fragment reassembly queues (IPQ), sockets, pipes, mbufs, file system mount points, processes, and vnodes. A blocking disposition is provided for mbuf, socket, and IPQ initialization; if a failure occurs during label allocation, the mbuf and socket allocator code will return a memory exhaustion failure to the consumer. Unlike most other kernel objects, memory to hold the mbuf label is not stored within the mbuf structure itself: instead, it is stored in an m_tag hung off the mbuf header m_tag chain. Tags to hold MAC data will be allocated only when policies requiring MAC labels on mbufs are present in the system. This permits improved network performance of the MAC Framework in scenarios where flexible access control is required, but where mbuf labeling is not.

Additional support for persistent label storage is provided by any file system supporting extended attributes, including UFS1 and UFS2; while policies can determine whether and how the attributes are bound to policy-specific labels, the Framework constructs transactions to read, write, and cache vnode labels on supporting file systems.

This flexibility supports a wide variety of behaviors required for many interesting and useful access control policies.

5.5 Composition

Hardened or trusted systems are frequently shipped with a number of active (and hence composed) security policies. For example, many traditional "trusted" UNIX systems include the standard UNIX access control model, local discretionary extensions to that model (such as ACLs), the Multi-Level Security (MLS) confidentiality model protecting user data, and the Biba integrity model protecting the integrity of the Trusted Code Base (TCB) [3] [4]. Likewise, locally maintained security extensions are frequently deployed in combination with existing system security policies, forming cohesive (and ideally stronger) protection. As the MAC Framework is intended to assist vendors in combining security components to be deployed in a variety of environments, the

MAC Framework supports the simultaneous loading of several modules. While it may not be possible to coherently (or even safely) compose all access control policies, the MAC Framework provides a simple composition model that has proven useful in existing shipped systems: rights intersection. This composition largely maintains and assumes independence between the active policies, composing their behaviors only for two classes of operations:

- *Access control checks.* A precedence operator composes the results of an access control decision; the practical impact of this approach is that if any policy denies access to an object or operation, then the MAC Framework will return an access denial to the kernel service. However, the precedence operator also has the effect of sorting "Object not found" errors before "Access denied" errors, providing some useful precedence behavior when information flow policies are present.
- *User label requests.* Policies are permitted to deny access to relabel objects even if the label change request pertains only to label elements maintained by other policies. This has utility in a number of situations, including in the following example: the Biba integrity policy may forbid the changing of an MLS sensitivity label on a high integrity object by a low integrity subject, since the changing of a label might constitute an information flow operation from the perspective of the Biba policy.

The same composition model is employed to combine results from the native UNIX access control model and any models added using the MAC Framework. Currently, the task of determining whether two policies may be safely composed is left to the system designer or administrator, a reasonable requirement for many of the deployed environments of interest.

5.6 Policy Modules

Policies are typically encapsulated in a kernel module, although they may also be directly linked to the kernel. Policy modules consist of several elements (some optional):

- Configuration. Optional configuration parameters for the policy.
- Policy logic. Optional abstracted and centralized implementation of the policy's access control logic.
- Labeling. Optional support for initializing, maintaining, and destroying labels on selected objects.
- Label APIs. Optional support for user process inspection and modification of labels on selected objects.
- Access control. Implementation of selected access control events that are of interest to the policy.
- Policy events. Optional implementation of policy initialization and destruction events.
- Declaration. Declaration of policy module identity, policy module properties, and registration of relevant policy operations.

5.7 Application Interfaces

The MAC Framework provides support for a number of classes of security-aware applications, including policy-agnostic or policy-aware labeling tools, and the login/user context management context routines. This is possible due to the policy-agnostic label management library and system calls, which allow applications to deal with MAC labels and elements in an abstract manner. The following functions are available to applications linked against the C library:

Retrieve the label of current or arbitrary process; set the current process label. `mac_get_pid()`, `mac_get_proc()`, `mac_set_proc()`.

Get and set file or pipe label by file descriptor. `mac_get_fd()`, `mac_set_fd()`.

Get and set file label by path; optionally follow symbolic links. `mac_get_file()`, `mac_set_file()`, `mac_get_link()`, `mac_set_link()`.

Execute a command and atomically modify the process label. `mac_execve()`.

Policy-specific system call multiplexor `mac_syscall()`.

Test for the presence of the MAC Framework or a specific policy `mac_is_present()`.

Convert labels to and from human-readable text `mac_from_text()`, `mac_to_text()`.

Allocate storage for a label appropriate to hold the specified label elements, or for a specific object based on system default label elements `mac_prepare()`, `mac_prepare_file_label()`, `mac_prepare_ifnet_label()`, `mac_prepare_process_label()`.

Release storage associated with a label `mac_free()`.

To support atomic change of label with execution events, `mac_execve()` provides an extension to the existing `execve()` system call accepting a requested target label. This is required to support the `execve_secure()` functionality used by the SEBSD port of FLASK/TE to FreeBSD from SELinux [11].

In addition, a general security policy entry point, `mac_security()` is provided so that policies may extend the set of system services without allocating new system call numbers.

6 Login Context Management

Many labeled access control policies assign user process labels on the basis of the identity of the user and properties of the user account. Frequently this is performed in a role-based manner, where a set of available roles is assigned to a user, and then the user may select their active role from among the available roles. This assumes the ability to assign an initial label during the login process or when acting on behalf of the user, and then the ability to support constrained modification of the label based on the initial login configuration. The MAC Framework user process labeling APIs are sufficiently flexible to support this behavior.

For an initial pass at supporting automatic labeling at login, we extended the existing BSD login class database. The `master.passwd(5)` assigns one class to each user; a class may be shared by many users, and includes information such as the resource restrictions for the user, login and accounting properties, etc. We introduced two new fields:

Identifier	Description
label	The text form of the label to be assigned to user processes as part of the context management process.
ttylabel	The label to assign to the user's tty

The existing `setusercontext(3)` interface is extended to support a new flag `LOGIN_SETMAC`, indicating that the MAC label should be set as part of the login process. This flag is also implied by the `LOGIN_SETALL` flag used widely across programs setting user contexts. Process labeling tools, described in the next section, may then be used to update the process label subject to policy constraints. By instrumenting this one function and its relevant consumers, we were able to easily modify most key system daemons and applications to recognize the new process properties, including sendmail(8), cron(8), login(1), su(8), ftpd(8), inetd(8) and others, making the changes relatively low impact. In the future, we may divorce the label selection database from the class database for the purpose of improved management, but this would not require changes to the user context API.

7 Application Integration

As most of the extensions policies of interest are mandatory policies, many applications that have specific adaptation to the system discretionary policy do not require changes for MAC. This occurs because objects created by processes will have labels automatically determined based on the process label or other process properties, rather than as application-provided arguments. The TrustedBSD MAC implementation ships with several tools to permit users to inspect and maintain labels on objects, including:

Program	Description
getpmac	Inspect process MAC labels
setpmac	Set process MAC labels
getfmac	Inspect file MAC labels
setfmac	Set file MAC labels
setfsmac	Set file MAC labels based on a specification file

In addition, the following utilities were also modified to inspect and set MAC labels:

Program	Description
ifconfig	Inspect and set interface labels
ps	Inspect process labels
ls	Inspect file labels

Further extensions could easily be made to applications such as the KDE file system browser, Konqueror, to display and manage labels on file system objects.

8 Sample Policies

FreeBSD 5.0 ships with a number of sample policies—many appropriate for deployment in production systems. These demonstrate some of the scope of the capabilities of the MAC Framework, ranging from very simple un-labeled inter-process visibility protections to fully labeled policy environments such as Type Enforcement.

- `mac_biba`. Fixed-label hierarchal Biba integrity policy with compartments: assigns integrity labels to all system subjects and objects, then enforces an information flow policy based on limiting read-down and write-up operations.
- `mac_bsdextended`. File system firewall, maintains an access control rule list expressed in terms of UNIX credentials, file owners, and operation masks.
- `mac_ifoff`. Interface silencing policy, prohibiting unauthorized output on network interfaces—appropriate for use in environments where silent monitoring is required.
- `mac_lomac`. Floating label hierarchal Biba integrity policy based on the "Low watermark" scheme [7]: assigns integrity labels to all system subjects and objects, preventing write-up and forcing a subject downgrade on read-down.
- `mac_mls`. Fixed-label hierarchal Multi-Level Security confidentiality policy with compartments: assigns sensitivity labels to all system subjects and objects, then enforces an information flow policy based on limiting write-down and read-up operations.

- mac_none. Stub policy providing prototypes for all policy entry points—a starting point for new policies. Also useful for raw performance measurements without the cost of labeling and access control events.
- mac_partition. Simple labeled system partitioning policy, in which processes are assigned to system partitions and visibility of processes is limited based on the label of a process.
- mac_portacl. Add access control lists to control explicit IPv4 and IPv6 socket binding by protocol, port, and uid or gid.
- mac_seeotheruids. Simple system partitioning policy, in which process visibility is limited based on the UNIX credential of a process.
- mac_test. Policy to exercise the MAC Framework as well as test its invariants. Checks to make sure the MAC Framework is correctly managing the labels on objects, and instrumenting appropriate access control checks.
- sebsd. Port of the SELinux FLASK and TE implementations to FreeBSD, providing access to the FLASK security abstractions, Type Enforcement implementation, and adaptations of a mature system policy.

A broad scope of policies may be implemented using the MAC Framework; the Framework is structured so that policy authors may select what performance, security, and functionality trade-offs they wish to make in policy design, augmenting the system policy in ways that reflect local requirements. This flexibility makes the MAC Framework a useful tool in a broad variety of environments, reflecting the variety of deployment scenarios in which FreeBSD is used.

9 Performance Results

Three important performance goals were kept in mind during the design and implementation process for the TrustedBSD MAC Framework:

- Minimize performance impact of the MAC Framework on systems where it is disabled.
- Minimize the overhead of the MAC Framework on systems where it is enabled and possibly in use.
- Permit policy authors to make performance/security/complexity trade-offs local to their policy based on the requirements for the policy.

In this section, we explore some of the issues associated with performance measurement of the MAC Framework. The Framework is currently integrated into the FreeBSD 5.0-CURRENT development branch—as a result, current performance measurements are used to guide the development process and explore the eventual impact, rather than representing final performance results. Substantial effort has not yet been invested in fine-grained performance tuning, although initial measurements suggest performance well within the bounds of acceptability.

For each test, we consider several kernel configurations:

- GENERIC: Base-line kernel without MAC support.
- MAC: Kernel compiled with MAC support, but no active security policies.
- MAC_NONE: One active "stub" policy, implementing all entry points but without additional locking or logic.
- MAC_BSDEXTENDED: One active "file system firewall" policy, implementing file system access control entry points and making use of a locked policy.
- MAC_BIBA: One active mandatory integrity policy, implementing comprehensive labeling and access control entry points for all system objects.

The GENERIC kernel permits us to explore baseline performance as a control for other configurations; MAC tests the overhead to simply include extensible security support in the system with no policies. The three sample policies allow us to consider the overhead of entering a policy module for each entry point (MAC_NONE), the cost of unlabeled file system protections using a locked policy (MAC_BSDEXTENDED), and the cost of a fully labeled system integrity policy touching most aspects of system operation (MAC_BIBA).

These tests were run on a FreeBSD 5.0-CURRENT system from the trustedbsd_mac development branch from late March, 2003; tests were run on a single-processor 800MHz Intel PIII system with 128mb of memory and ATA 7200rpm 20gb hard disk. For file system related benchmarking, all writable file systems were recreated using the same geometry between tests since file system aging effects are not of interest for these tests; reboots occur between each test to flush storage-related caches and reset slab allocator and mbuf allocator state. All file systems use UFS2 for high performance meta-data storage.

9.1 Kernel Compile Throughput

In the buildkernel test, we perform a macro-benchmark focused on system throughput relying on effective CPU utilization, I/O performance, and file system meta-data performance. In this test, a FreeBSD kernel source tree is configured and built without modules (to reduce the I/O throughput dependency); time is measured in wall clock duration from start to finish. Lower execution times are preferred, indicating higher system throughput in completing the task.

The results of this test demonstrate a small but measurable performance change (0.1%) with MAC support. A slight relative increase in cost for the BSD/extended policy may be the result of acquiring a policy lock in order to process an access control decision; however, there is no statistically significant difference in performance between the various MAC policies and the base cost of the MAC Framework; UFS2 provides for high performance label access for the Biba policy.

9.2 Network Performance

The MAC Framework introduces security label structures into a variety of system data structures; of these, `struct mbuf` may be the most performance-sensitive. The `mbuf` structure provides for optimized network management, and has been the subject of substantial prior performance work, providing optimized packet construction and parsing, copy-on-write semantics, zero-copy semantics, and fragmentation management. From the perspective of MAC policy modules, only header `mbufs` are of interest, as they represent the header for a network packet or datagram. In our first pass implementation, we inserted a `struct label` directly into the `m_pkthdr` data structure; as the implementation evolved, the `m_tag` meta-data service became available on FreeBSD; this service permits chaining of arbitrary meta-data onto `mbuf` headers without modification of the base structure.

In the `m_pkthdr` approach, all kernels pay a memory overhead for labeling support, although kernels without `options MAC` do not pay the label life cycle costs. With the `m_tag` approach, only kernels with `options MAC` pay the memory overhead, although we presupposed that there would be a higher cost for using tags for label storage due to greater administrative overhead in maintaining lists and allocating storage. To optimize the `m_tag` approach, we implemented lazy tag allocation: tags are only allocated to hold label data when a policy expresses interest in labeling `mbuf` headers.

We consider two tests from the `netperf` suite: `UDP_RR` and `UDP_STREAM`, which respectively test the per-transaction cost of a Request/Receive RPC, and raw network throughput. The request/response test measures the throughput of the system relative to synchronous one-byte packets between a client and a server, and is intended to measure the performance impact of a change in terms of number of packets transfered. The stream test uses a larger packet size and does not synchronously wait for a response before continuing, generally measuring the performance impact of a change in terms of data transferred.

In Figure 5, the performance cost per-packet is illustrated: the introduction of MAC support produces a measurable change; depending on the strategy for labeling mbufs, that change varies substantially. With inclusion of the label directly in the mbuf header, an 11.5% performance overhead is accepted for enabling MAC support. Adding the stub policy increases that cost to 12.2%; adding a complex labeled policy performing per-label memory allocation, such as Biba, increases that drop to 14.9% of the GENERIC packet throughput.

With lazy m_tag labeling, the performance trade-off is changed: the cost of introducing MAC is 4.8% (substantially less than using mbuf headers); with a stub policy implementing mbuf label entry points but not allocating labels, that cost increases to 8.5% (also less than mbuf headers). However, performance with a Biba performance is reduced by 17.1%, showing an increased cost for heavily labeled policies such as Biba.

The second set of trade-offs best fits the needs of the TrustedBSD Project: minimize overhead for MAC-disabled systems, and permit a performance/complexity cost decision by policy authors.

In Figure 6, a similar pattern emerges, where-in the introduction of MAC support results in a 6.1% throughput penalty. Use of a stub policy increases that cost to 7.7%; Biba labeling increases the cost to 10.3%. However, with lazy m_tag labeling, the base cost of MAC support is reduced to 3.7%; the stub policy increases this cost to 4.4%, and with Biba to 9.7%. The proportionally lower performance cost with this test derives from the reduced relative overhead resulting from reduced packet counts relative to data transfered.

Again, lazy m_tag allocation better meets our requirements by permitting better non-MAC performance with a more clear performance trade-off for complexity. Unlike the packet count testing, performance for the Biba policy actually increases with lazy m_tags relative to mbuf headers, a result that we attribute to differences in the caching and allocation policies for the UMA Slab Allocator versus the mbuf allocator in handling memory clearing for new allocations.

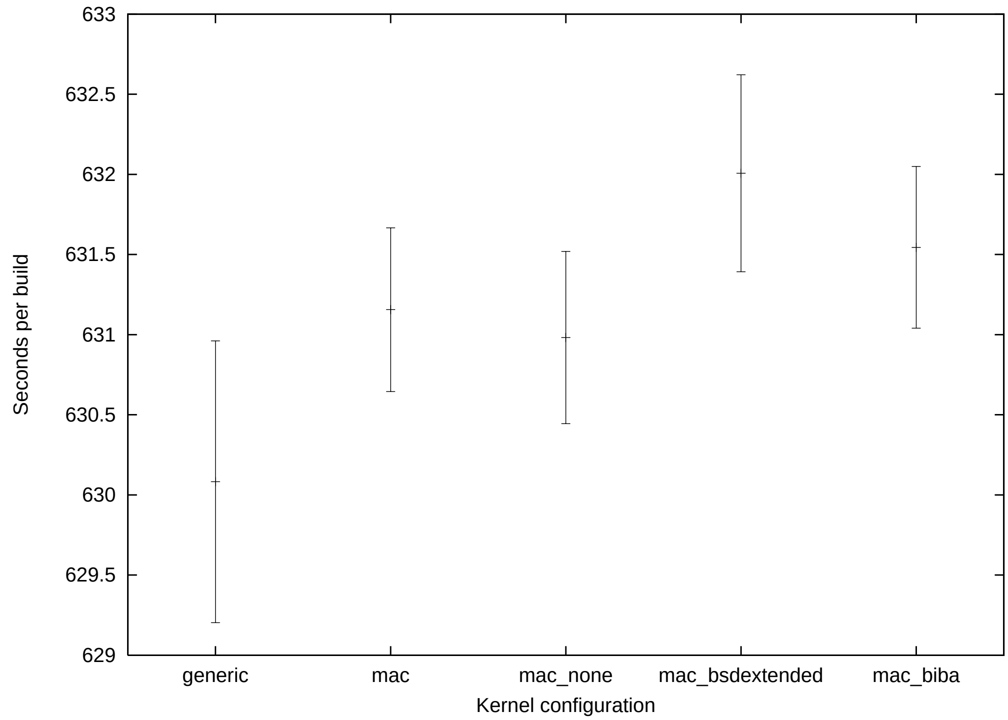

Figure 4: Time to `make buildkernel` with various kernel configurations (lower is better).

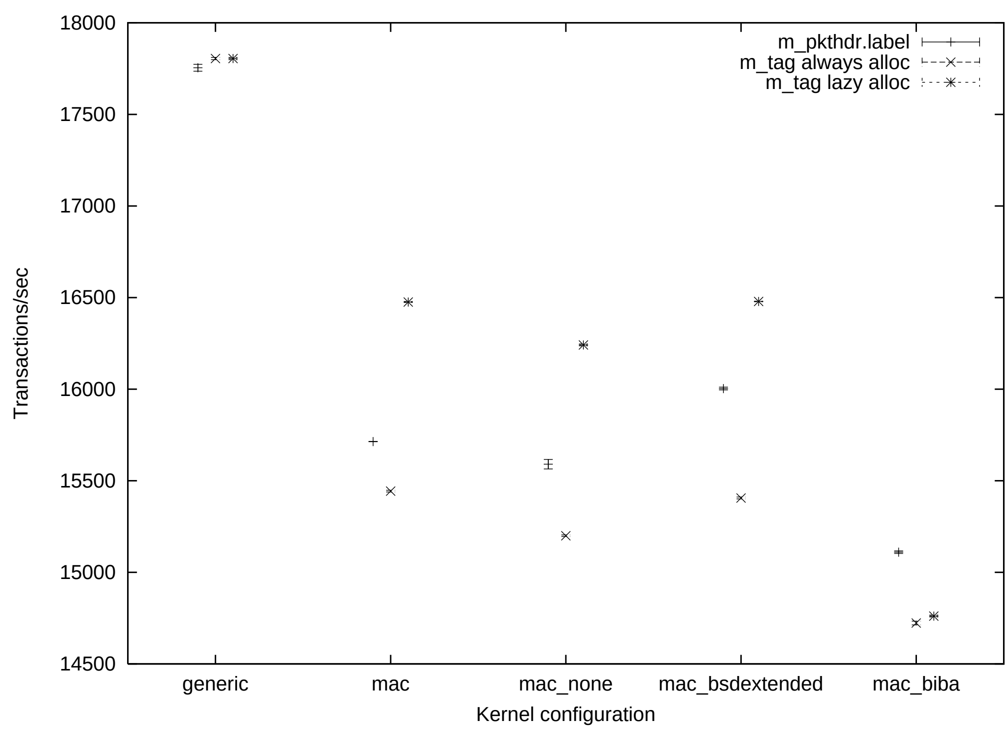

Figure 5: UDP request/response throughput over loopback with various mbuf label allocation approaches (higher is better).

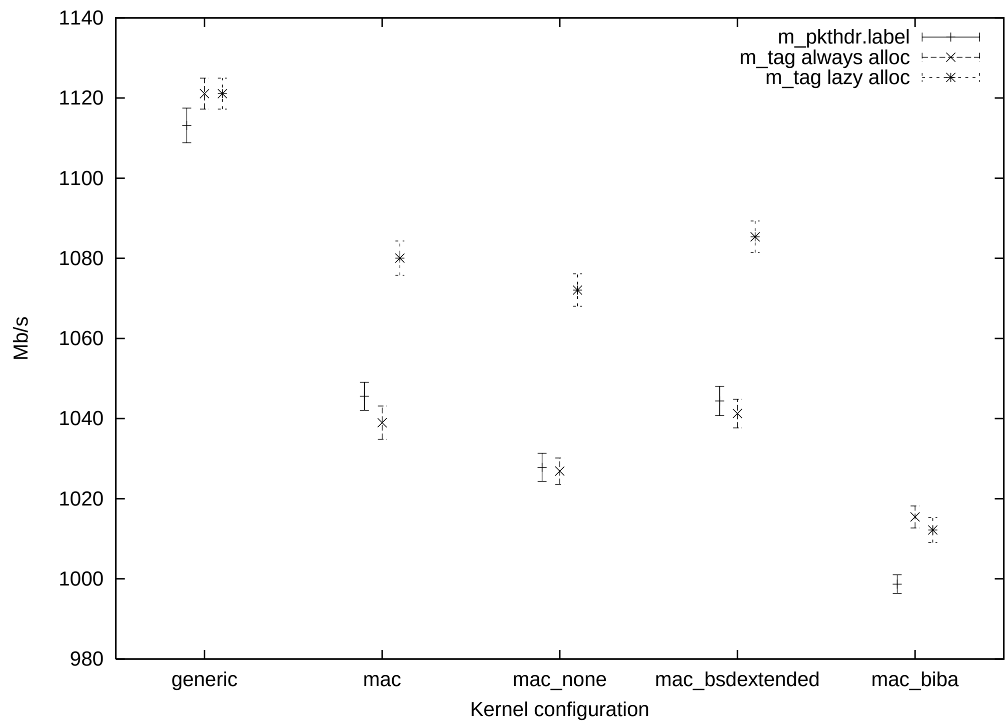

Figure 6: UDP stream throughput over loopback with various mbuf label allocation approaches (higher is better).

10 Related Work

Substantial prior and current work exists relating to kernel access kernel and extensibility research.

In the area of prior deployed systems, "Trusted" variants of most commercial UNIX platforms exist, including Trusted Solaris, and Trusted IRIX [14]. In addition, there are a number of third party security extension products that exist for these systems, including Argus's PitBull product, which provides a product alternative with many of the same features [2]. These products largely rely on Multi-Level Security (MLS) [3] and Biba integrity [4] to provide mandatory data confidentiality and TCB integrity; earlier TrustedBSD work has focused on implementing support in FreeBSD for these models using similar integration approaches [17][18][19]. TrustedBSD MAC policy modules exist expressing both MLS and Biba in functionally similar forms.

In the area of access control extensibility, early work included the Generalized Framework for Access Control (GFAC), which proposes a separation of policy and enforcement [1]; this model is implemented in Linux in RSBAC [13].

The FLASK framework provides for similar types of separation of policy and enforcement, although with higher level labeling abstraction in the form of a security ID (SID) and a focus on Linux Security Modules (LSM) provides a set of kernel extension hooks to facilitate integration of systems such as SELinux without committing the Linux operating system to a particular model [16]. LSM provides a void pointer for label storage in each supported kernel object, and has access control notions similar to the TrustedBSD MAC Framework. However, the semantics of the hooks are weaker, and the LSM framework does not provide for policy composition and persistent labeling, relying on policy modules to implement these services. Type Enforcement for policy representation [15][11]. A prototype port of the SELinux FLASK and TE implementations has been made to layer on top of the TrustedBSD MAC Framework via the SEBSD policy module.

11 Future Work

Future work on the MAC Framework will likely fall into a number of areas:

- Improve the completeness and expressiveness of the MAC Framework; increase the number of kernel objects and methods that are protected by the Framework to permit broader protections.
- Mature the experimental policy modules.
- Continue to adapt and merge the SEBSD policy module to run properly with FreeBSD: in particular, determine how best to satisfy the differing requirements of SEBSD and most other policies regarding process label transitions.
- Continue porting the MAC Framework and its policies to Darwin and Mac OS X.

12 Conclusion

The TrustedBSD MAC Framework provides a generalized mechanism by which the FreeBSD kernel security model can be augmented at run-time. Along with the framework, we have also implemented a number of security extension modules that rely solely on the framework to interface with existing kernel abstractions. This separation of security extensions from the actual kernel implementation of services improves the capacity for third party providers to develop and ship system security extensions by lowering the cost to develop and maintain the extensions. Through a simple composition model, it is possible to perform a limited set of "useful" compositions of security extensions. Preliminary performance measurement illustrates a measurable but small performance cost for the framework and many policy modules. the Framework permits policy authors to select complexity and performance trade-offs based on local requirements, supporting both simple hardening policies and complex information flow policies.

13 Acknowledgments

This research was supported under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS").

The authors would like also to thank the anonymous reviewers of this paper, as well as other contributors to the TrustedBSD Project, including Chris Faulhaber, Ilmar Habibulin, Adam Migus, and Thomas Moestl. The authors would also like to thank Angelos Keromytis and Erez Zadok for their shepherding of this paper.

14 Availability

The TrustedBSD MAC Framework is available under a two-clause BSD license, making it appropriate for open and closed-source, research, educational or commercial use without restriction. It is included in FreeBSD 5.0, as an experimental feature, and will mature over the FreeBSD 5.x life time. More information may be found at:

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

15 Bibliography

References

[1] 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*, pages 135–143, 1990.

[2] Argus products overview: Pitbull. http://www.argussystems.com/product/overview/pitbull/.

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

[4] K. Biba. Integrity constraints for secure computer systems, 1977.

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

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

[7] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[8] FreeBSD Project. FreeBSD home page. http://www.FreeBSD.org/.

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

[10] N. S. A. Information Systems Security Organization. Labeled security protection profile version 1.b, October 1999.

[11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, U.S. National Security Agency (NSA), Oct. 2000.

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

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

[14] SGI. B1 sample source code. http://oss.sgi.com/projects/ob1/.

[15] 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*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX.

[16] G. S. Support. Linux security modules:.

[17] TrustedBSD Project. TrustedBSD home page. http://www.TrustedBSD.org/.

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

[19] R. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.