TrustedBSD
Adding Trusted Operating System Features to FreeBSD

Robert N. M. Watson
FreeBSD Project, NAI Labs
rwatson@{FreeBSD.org,tislabs.com}

Abstract

Trusted operating systems provide a "next level" of system security, offering both new security features and higher assurance that they are properly implemented. TrustedBSD is an on-going project to integrate a number of trusted OS features into the open source FreeBSD operating system, and involves both architectural and development process improvements. This paper describes how the open source development practices of the FreeBSD Project impacted the design and implementation choices for these features, and describes lessons learned that will influence future work. Several key TrustedBSD features are discussed as examples of how new security services may be introduced in such an environment.

1 Introduction

TrustedBSD[20][22] is a project to add trusted operating system functionality to FreeBSD[7][14], including improvements to the kernel and userland security infrastructure, services to better support security features, and specific security features including Access Control Lists (ACLs), fine-grained privileges ("Capabilities"), and Mandatory Access Control (MAC). While TrustedBSD is still under development, several features are already complete and now integrated into the base FreeBSD distribution for inclusion in FreeBSD 5.0. This process has resulted in a great deal of gained experience, which in turn can be used to draw useful conclusions about how future work, especially with regards to development practices, should be performed. The integration of new security features in FreeBSD offers a number of practical lessons, both technical and social.

This paper introduces the basic features that make up TrustedBSD, describes the goals and processes by which these are being accomplished, details a subset of the features, and reflects on lessons learned as well as future directions for work.

2 TrustedBSD Feature Set

The TrustedBSD feature set attempts to address a number of requirements originating from several communities. This includes the traditional trusted operating system community, in the form of the Orange Book[16], and more recently the Common Criteria[5][11][10], but also the more widespread desire in the FreeBSD community for improved security functionality in the form of greater protection flexibility. These features improve both improvements in existing code, development process improvements, and specific new features.

- *Improved security consistency and correctness.* An important part of introducing new security models and extensively modifying security subsystems is verifying that they are correctly implemented. The FreeBSD developer community did not have test suites covering security behavior, so tests are being developed to determine that no new vulnerabilities are introduced, and so that changes can be experimentally quantified.

An early observation made during this process was that supposedly-equivalent security checks would often be implemented differently. For example, different access control checks were used for the two ways in which debugging can be attached to a process, `ptrace()` and the process file system. Little or no code sharing, combined with an incremental development style, has lead to inconsistent and undocumented protection behavior, especially in the areas of inter-process authorization and file system permission evaluation.

Correctness is clearly an important part in any security project; however, without appropriate tools to verify correctness, it can be difficult to achieve. Access control consistency (and hence abstraction), careful documentation, and extensive and rigorous testing are necessary to accomplish this goal.

- *Improved security abstractions and modularity.* Introducing common implementations of access control check code across check instances improves consistency, but also allows the introduction of improved abstractions, making it easier to introduce new models by substituting policy logic at well-defined enforcement points. Likewise, improving the abstractions associated with the labeling of subjects (process credentials) and system objects (such as files, network packets and interfaces, and kernel management services) allows the more consistent introduction of comprehensive security features such as many MAC policies.

- *General services to support security requirements.* A number of the components of TrustedBSD have usefulness outside of the base TrustedBSD feature set, providing utility for purposes other than purely security work. One example of this is in file system Extended Attributes (EAs), which provide a general interface and implementation for associating arbitrary meta-data with files and directories. A number of the new security services require that additional security labels be associated with file system objects; however, EAs can also be used by applications to store version metadata, portability information, or even file icons.

- *Fine-grained Discretionary Access Control (DAC).* The UNIX permission model allows users to specify discretionary protections for objects they have created; this mechanism, however, is inflexible and inexpressive, particular in large-scale environments, or where there are complex security requirements.

  POSIX.1e[9] Access Control Lists (ACLs) allow object owners to specify finer granularity protections for file system objects. The ACL implementation leverages the availability of a general EA service, and provides a high level of compatibility with the permission model. ACLs are not only a relatively simple implementation task, but are also a "bullet feature" expected by many operating system consumers, making them a particularly appealing target.

- *Fine-grained privilege model.* One traditional criticism of the UNIX security model has been its reliance on a single concentration of system privileges in the "root" user. This focuses an unnecessarily high level of privilege in a large number of applications, violating the principle of least privilege, and leaving the applications as easy targets for attackers.

  POSIX.1e capabilities decompose the root privilege set into several logical components, decoupling privilege from the UID of the process. Processes may manage the availability, inheritance, and effectiveness of capabilities, limiting the scope of damage due to compromise. This implementation leverages EAs to bind capabilities to binaries, and improved security abstractions to replace the superuser access control checks.

- *Mandatory Access Control (MAC).* MAC permits security administrators to define mandatory security policies regarding the relationships between subjects and objects in the system. Traditional MAC policies have included Multi-Level Security (MLS)[3] which provides a military-style confidentiality policy, as well as a variety of integrity and safety models such as Biba integrity models[4], compartmentalization models, and more general policy mechanisms such as TE[13] and DTE[2].

Free UNIX-like systems have traditionally lacked such features, which can provide higher levels of protection; mandatory policy enforcement is one of the determining features associated with the traditional trusted operating system. Many of the enforcement points for MAC already exist in FreeBSD by virtue of the existing security models, including the Jail[12] model, but improved labeling and access control abstractions, as well as the ability to store labels persistently in EAs, are required for most MAC policies.

- *Plugability.* A long-term architectural goal is to allow the rapid introduction of new security services by continuing to improve abstractions and encouraging modularity. An important element of this is untangling the existing set of security models into independently structured components which are then cleanly composed to generate access control decisions. Providing an easy means to introduce and integrate new models will promote the development of new security features by simplifying the development process. This goal presents substantial challenges, both to design and implement, and in maintaining the necessary performance and usability characteristics.

- *Documentation and Education.* The TrustedBSD feature set introduces a large number of new interfaces and services that are relevant to both system developers and users. A substantial and continuing effort is being invested in maintaining up-to-date developer documentation, and as features are integrated back into the base FreeBSD distribution, user documentation must also be brought up-to-date. Trusted operating system features are a topic unfamiliar to many system programmers and users: detailed yet readable documentation is vital to both maintaining the correctness of the implementation, and to introducing operating system consumers to the suite of newly available functionality.

3 Implementation and Process Goals

FreeBSD is an actively developed and widely-deployed high performance production operating system. As a result, introducing new security features into the base distribution places a number of constraints on their implementation. This includes the desire to introduce features that provide a rapid security benefit, to avoid degrading performance in existing deployed configurations, to introduce features in a manner compatible with the FreeBSD development and release schedule, and to participate in on-going education and public relations to introduce and promote the ideas and features within the FreeBSD community. In addition, outreach to other operating system communities is necessary to develop standards for application interfaces so as to assure portability of applications taking advantage of these features.

3.1 Satisfying the Requirements of a General-purpose Operating System

- *Rapid security improvement.* Target improvements that have a demonstrable security impact in the short term without high development cost, such as improved access control consistency, code sharing, correctness checking. These changes produce real-world security improvements, as well as making it easier and safer for developers to integrate new security models.

- *Popular features first.* Target features that are more generally useful first, especially primitives that may be reused. This includes extended attributes, ACLs, and improvements to existing security services such as the `jail()` code. Some features, such as MAC, offer benefits but require a substantially higher investment, as well as further research into how they can be deployed in existing environments, and should be considered long-term goals.

- *Minimize cost on today's deployed installations.* Initially optimize for minimal impact on currently deployed configurations (where new security features will not be enabled), maintaining performance and ease of use. Otherwise, the community will resist the efforts to introduce new features, as they would be contrary to stated goals of the FreeBSD project. Integrating slower features into the base distribution as optional components will, however, increase exposure in the broader developer community increasing the chances of additional developers picking up and working on the implementation to improve perceived performance problems.

- *Support the applications.* Provide security services that allow as many existing applications to run as possible. In practice, this has two implications: first, applications unaware of security mechanisms must behave correctly, and if they must fail, do so safely, and second, security-aware applications must continue to function properly, possibly adapting to support new security primitives. This strategy encourages the adoption of new security features while avoiding introducing risks by changing the fundamental assumptions of application interfaces (in particular, POSIX).

3.2 Emphasis on Portability

Each TrustedBSD feature has introduced a plethora of new APIs for providing access to new services. If similar services exist on other platforms, it is desirable that applications written on one platform be portable to the other. This will be possible only through close communication and cooperation with other vendors, and in some cases, through the development of new standards.

The starting point for this work has been POSIX.1e, a withdrawn IEEE specification draft intended to provide portable interfaces for Access Control Lists, Auditing, Capabilities, Information Labeling, and Mandatory Access Control. As many of these topics are contentious within the security community, large parts of the draft are effectively unusable as they constitute a consensus on the need for a feature, rather than practical interface details needed for actual implementation. However, the ACL and Capability components of the draft are quite usable, with partial implementations of both widespread. We selected Draft 17, the final draft of the specification, as a starting point. We made extensions or modifications where necessary to disambiguate aspects of the draft, provide functionality not anticipated by the draft writers, or to handle non-POSIX and BSD-specific extensions.

POSIX.1e does not describe extended attributes (EAs), although a number of POSIX.1e implementations rely on EAs to provide storage to support its features. This includes SGI's Trusted Irix[18], FreeBSD, and now also Linux[8]. As EAs will likely be consumed by applications directly, as well as by kernel security services, adopting consistent application interface syntax and semantics is highly desirable. The POSIX.1e online discussion mailing list has provided a forum for the discussion of EA interfaces; a final interface has not been agreed upon, but there is a reasonable consensus on the desired semantics.

The mandatory access control interface described in POSIX.1e, on the other hand, may be too specific to the MLS and Biba MAC models, which each define a dominance operator, requiring a policy that orders labels. The interface also lacks a means by which user processes can host objects and enforcement points, but rely on the operating system to provide label management and policy service. There is substantial consensus in the broader community that more general access control primitives are required to support a broad array of flexible policy mechanisms, and that the POSIX.1e interfaces may provide a useful starting point for that work.

Working with existing models, where possible, offers substantial benefits in the form of application portability. It also allows for a faster design and implementation process, as there is greater understanding of the model (including its limitations), reducing development risk. Where portability standards do not exist, it is desirable to develop new standards, such as with EAs. Creating many divergent "Trusted Sendmail" implementations to account for many MAC interfaces, for example, is clearly undesirable, both from the perspective of increased workload, risk associated with reduced review, and divergent (and conflicting) security properties.

3.3 Gradual Integration via the Open-source Approach

The open source development processes differ substantially from many commercial development approaches. The distributed volunteer-oriented development process reinforces a number of design and development trends, resulting in a rapid development cycle, featurism, integration of experimental features, and diverse models of "success".

For many open source projects, the motivations for developers are different from those of closed source commercial products: they are highly motivated to do the work, but often have limited resources to bring about the results they seek. This can result in a "many testers but few developers" syndrome for features that are either less popular or technically difficult to implement. The volunteer nature of the work means that the model of success is often based on the degree to which the software is available and used, and the effectiveness in attracting new developers to a project, rather than monetary compensation.

The limitations of version control and collaboration tools often drive the organization of open source software projects that use them. For example, CVS's inability to effectively handle a three-tier development process (central repository, per-project repository, local development tree) makes it difficult to track the rapidly moving central FreeBSD source repository without pushing changes back into the central source tree. This further encourages the wide-spread open source technique of providing early access to work still under development, allowing for broader exposure of the code and therefore more effective testing. Providing early access to the EA implementation greatly facilitated the development of TrustedBSD features by permitting independent development of other features, otherwise made difficult by CVS's inability to handle a hierarchal model. Likewise, allowing early access to the ACL implementation, even though it was still partially complete, allowed for far broader testing and greater numbers of developers.

Releasing early and often during the development process often means submitting the necessary hooks to support easier development, such as reserving system call numbers and adding prototype interfaces. These techniques are appropriate where hooks and interfaces are intended to remain relatively static, but allow the feature under development to generate few modification conflicts even as the base tree moves forward. For example, a number of the TrustedBSD APIs appeared in the 4.x-STABLE FreeBSD release branch, although the underlying implementations were not present. The development of improved abstractions and modular service interfaces allows the development process to be further streamlined—as better abstractions are introduced, the changes to the base source distribution necessary to support new features get progressively smaller.

The open source development process also allows a new element to be introduced in the software portability process: direct code sharing to improve interface portability. This facilitates the development of parallel implementations in a number of ways: the code may be directly "borrowed" from another distribution if the licenses are compatible, direct inspection of parallel code can improve consistency and correctness, and it is possible to take advantage of the source code for third party tools relying on the service to perform testing. The TrustedBSD project has frequently made use of open access to other systems' source to understand the interfaces and implementation quirks of services on those systems. Implementing ACLs, for example, was greatly facilitated by the ability to recompile and test the Linux `getfacl` and `setfacl` tools on FreeBSD to determine that they behaved consistently with the FreeBSD implementations, and that our ACL library routines behaved correctly.

For the TrustedBSD Project to succeed, it must leverage the benefits of the open source model while avoiding the pitfalls: in general, this means adapting the development cycle and processes to that of the FreeBSD Project, which has shown remarkable success in navigating the challenges of distributed collaboration and development. Understanding the social aspects of open source software development is also important, including accepting the open source success model, leveraging distributed development and testing, and using open source as a tool for improved portability.

4 Case Studies in Feature Implementation

These design and implementation goals outline a strategy for the development and integration of new advanced security features into the FreeBSD operating system. We now consider a number of the features under development as part of the TrustedBSD project, and how these goals have influenced the design and development process.

4.1 Regression tests

One of the primary challenges and risks of security feature development is that of correctness: unlike many areas of software authoring where it is possible to accurately capture the common case and then gradually address the less visited code paths, a single failure in security software can render the entire construction useless. This is especially true when introducing complex security features such as mandatory access control, where many components act in concert over a spectrum of system abstractions. As such, an important tool for successfully developing security features is a comprehensive set of tests and evaluation mechanisms, which can be used to analyze and quantify the existing implementation, then to perform incremental verification of any changes made. Part of the challenge also lies in that FreeBSD is not a perfect starting point: the existing implementation suffered from a number of inconsistencies which had to be understood—often these existed precisely because such tools were not available.

TrustedBSD testing tools fit into two general categories: tests intended to evaluate the correctness of specific aspects of the implementation, and tests intended to evaluate the overall correctness of larger scenarios. Smaller context-specific tests attempt to exhaustively explore the behavior of a specific piece of access control or related security functionality by constructing the relevant characteristic arguments and context, then comparing the results of the function with declared expectations.

An example of this includes the `proc_to_proc` regression test, which was developed to explore the correctness of authorization policy for inter-process system calls. Inter-process calls typically involve two processes: the first (subject) process invokes a system call which will affect another (object) process. Depending on the credentials associated with processes, and the security model in use, the kernel should reject some calls, and accept others. For example, the `ptrace()` system call allows a process to attach debugging services to another process, permitting it to read and write the memory contents and state of the process, as well as control its execution flow. Such a service allow the subject process to gain access to any resources available to object process, and as such, constitutes a substantial security risk if not properly protected. The following sample output from `proc_to_proc` illustrates a test failure when a process successfully signals another process instead of receiving the `EPERM` error.

```
[21. unpriv1 on daemon1].signal: expected
     EPERM, got 0
(e:1000 r:1000 s:1000 P_SUGID:0)
(e:1000 r:0 s:0 P_SUGID:1)
```

Larger scenario tests attempt to explore whether more general expectations for correct behavior are met by the system. These tests typically perform compound operations, checking only that, given the correct starting state and sequence of operations, the desired end property is present. For example, the `setuid_protected` test evaluates whether or not a process that has executed a setuid binary undergoes the expected credential transformation, and, if so, is then protected from manipulation by other processes present in the system.

In both types of test, a clear notion of "correct" and an understanding of potential failure modes is required to design useful and complete tests. This is not a challenge unique to informal regression testing of security functionality on open source operating systems, but is complicated by a lack of clarity as to what the intended model should be.

The regression test design and implementation task offers substantial benefits to both TrustedBSD developers, and to the broader FreeBSD community. The test suites have already been used to simplify a number of access control checks, as well as point out inconsistencies in access control implementation. By using these tests in the development process, it is possible to gain greater assurance that the new features being added are implemented correctly, and that they do not weaken existing protections.

4.2 Extended Attributes

Extended attributes (EAs) provide a clean abstraction for associating additional metadata with files and directories, a requirement for the implementation of many new kernel security features (ACLs, Capabilities, MAC, ...) as well as a feature which non-security applications may find useful. Providing a metadata storage abstraction reduces the implementation overhead associated with these features, both in terms of redundant work and the substantial complexity of extending on-disk storage formats. Specifying a clean API allows new services to be implemented without knowledge of underlying format, permitting a rapid EA implementation early in the project to facilitate development of a number of other features.

The EA interface provides simple semantics: for each file or directory, zero or more names may be defined. EA names exist in disjoint namespaces, of which two are defined: `EXTATTR_NAMESPACE_SYSTEM` and `EXTATTR_NAMESPACE_USER`. Namespaces determines the protection properties of an EA—access to the system namespace is limited to the kernel and privileged processes, while EAs in the user namespace are protected using the discretionary and mandatory protections on the file or directory. Each defined name may have zero or more bytes of data associated with it. No EAs are defined for a newly created file or directory, although consumers of EAs may define names and values during the creation process. Two operations are defined, allowing EAs to be atomically retrieved and set.

For a first implementation, we selected a simple design that permitted us to move on to additional new features that rely on EAs, allowing later performance optimization by those with greater expertise in file systems. Rather than modify the on-disk file system format, we chose to store EA data in backing files. This allowed us to avoid a lengthy and bug-prone development process, avoid conflicts with other on-going development on FFS, and avoid requiring low-level file system modifications to allow developers and users to experiment with EAs or features that rely on them. Each backing file stores one named EA from a single namespace for all files in the file system, and is treated as an array of EA instances indexed by inode number. Both the file itself and each instance of an EA have headers. The file header contains a backing file format version, as well as a field defining maximum size any EA instance can take on, permitting the array record size to be calculated as the sum of the EA instance header size and maximum EA instance size. EA instance headers indicate whether or not the instance is defined for the given inode, the size of the EA instance if defined, as well as a copy of the inode generation number, used for synchronization purposes. A privileged user process can invoke the `extattrctl()` system call to start EA support on a given UFS-based file system, and then enable individual EAs by

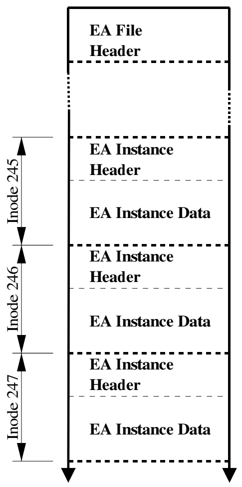

Figure 1: EA backing file format

associating backing files with EA names and namespaces.

It is also possible to have EAs automatically started and enabled for the file system at mount-time by specifying the `UFS_EXTATTR_AUTOSTART` kernel option. When enabled, the mount code will search the `.attribute/system` and `.attribute/user` directories off of the file system root for valid backing files. When a file is found, an EA with the same name is enabled in the appropriate namespace. This permits atomic starting of EA services with the mount operation, preventing race conditions that might be present as a result of a delay in EAs becoming available while other files in the file system are accessible.

This implementation offers acceptable performance, requiring an additional seek for most operations if the EA has not already been loaded from disk. Currently, the UFS EA implementation relies on the file system buffer cache to cache the backing file, rather than implementing a custom EA cache; the temporal locality properties of most services currently layered on EAs allow this caching to be effective in mitigating most performance costs.

This implementation is sufficient to implement services such as ACLs, Capabilities, and MAC above the EA interface. However, it suffers from a number of limitations, including the treatment of EA meta-data as "data" from the perspective of the file system synchronization policy, in particular, with regards to the soft updates mechanism used in FFS. One important synchronization failure mode occurs if an EA is not always enabled when the file system is active. In this scenario, two problems arise: first, EAs are not garbage collected at file deletion, and second, services relying on EAs cannot update meta-data. The inode generation number replication into EA instance headers permits some synchronization problems to be detected, by preventing old EA data from being used with a new file, as the inode generation number is changed when the inode is re-allocated. In large part, the service meta-data update problem is solved by allowing the atomic auto-starting of EAs at mount-time. Currently, work is in the planning stages for a block-level implementation in FFS, which would have stronger performance and consistency properties while retaining the same interface, requiring no change to services above it.

4.3 Access Control Lists

Access Control Lists (ACLs) allow users to express more detailed policies for files and directories that they own. POSIX.1e defines an ACL interface that acts as a superset to the current permission mechanism: the file access ACL consists of a base ACL derived from the file permissions, and the extended ACL defines permissions for additional users, groups, and an optional ACL mask. Each non-mask entry in the ACL associates a user or group with a set of rights that the user or group will have on the file or directory.

The ACL evaluation algorithm selects an appropriate part of the credential and an entry in the ACL that are combined during permission evaluation; this order of preference matches first the owner, then additional user entries, then group entries, and finally, the "other" entry. The POSIX.1e ACL mask plays an important role in providing compatibility for ACL-unaware programs: it places a bound on the maximum rights provided by any additional users or group entries. If an extended ACL is available for an inode, the `chmod()` operation on the file is modified: rather than setting the file group bits, the ACL mask is modified. As a result, modification of the group bits in the permission effectively masks the rights for all entries of the ACL other than the file owner and other entries, allowing programs not aware of ACL interfaces to place an upper bound on file accessibility. Additional compatibility is provided by a default ACL placed on directories, which is combined with the permission set provided by the process on `open()` or `create()` to produce the new access ACL for a file created in that directory, allowing ACL-unaware applications to create a new file with an appropriate ACL.

The FreeBSD implementation splits the ACL data over the existing inode mode field in UFS, and two EAs, `posix1e.acl_access` for the access ACL on an inode, and `posix1e.acl_default` for the default ACL. At the VFS layer, two new vnode operations are introduced: `VOP_GETACL` to retrieve available ACLs from a vnode, and `VOP_SETACL` to set ACLs on the vnode. The caller may specify the ACL type determining whether the ACL operation is intended for the access or default ACL. When ACL support is compiled into the kernel, ACL code is enabled in a number of other UFS vnode operations, including `VOP_ACCESS` which invokes a generic `vaccess_acl_posix1e()` access check routine, as well as during file and directory creation via `VOP_CREATE()`, `VOP_MKNOD()`, `VOP_MKDIR()`, and `VOP_SYMLINK()`, where the default ACL, if any, is combined with the requested file mode to produce the access ACL for the child. As the ACL is split over both the inode mode and EA storage, the fields must be synchronized during certain operations—in particular, the ACL vnode operations, but also during file creation to combine the default ACL and request mode.

As a result of splitting the access ACL in this manner, many frequently performed operations, such as `stat()` and `chmod()` incur no additional overhead. The access ACL must be read for `open()` and `access()` calls on a file, and during actual ACL read or update operations. Access ACLs impose a slightly higher cost on directory operations than on file operations, although they also exhibit higher locality: directory lookup and listing requires that the access ACL be evaluated for the `ACL_EXECUTE` and `ACL_READ` permissions, respectively. Creation of a new file or sub-directory within a directory also exhibits higher cost because both the access and default ACLs must be retrieved for the parent, and then new access and default ACLs may be written out for the child.

In practice, ACL operations have high temporal locality lending them to caching, and suffer from higher latency rather than actual disk I/O utilization increase. When ACLs are not enabled on the file system, there is no measurable performance difference from the pre-ACL implementation, in keeping with the "minimal impact on current configurations" mandate. When ACLs are enabled but not used, an overhead is perceived due to reads associated with determining if an access ACL is defined, and for the lookup of default ACLs during file or sub-directory creation. When ACLs are enabled and utilized, higher costs are perceived during file and sub-directory creation if a default ACL is set on the directory in which new children are created. To improve the actual cost of ACLs when in use, the primary target for optimization is the EA implementation: the measured costs of ACL operations is effectively identical to the measured cost of the EA operation supporting the ACL operation.

The POSIX.1e ACL specification offers a largely complete and unambiguous specification for an ACL implementation; some extensions, however, are required to add more complete functionality in FreeBSD, such as the ability to perform ACL operations on directories via a file handle. Although the ACL mask behavior increases complexity, it provides relatively transparent support for ACL-unaware applications. While the ACL specification is not identical to the variations used in many commercial UNIX variants, it offers compatible semantics. The ACL implementation will be included in FreeBSD 5.0-RELEASE, and a number of applications, including Samba, already work properly with ACLs on FreeBSD 5.0-CURRENT development branch.

4.4 Mandatory Access Control (MAC)

Mandatory access control permits security administrators to specify fine-grained policies limiting the interactions between users and objects on the system, which will be enforced regardless of the any permissions granted by discretionary access control primitives. Often, mandatory access control policies consist of schemas for limiting information flow, such as MLS and Biba policies, but may also consist of more general policies, such as Type Enforcement, or more specific policies, such as the FreeBSD jail mechanism. MAC policies generally require that subjects and objects be labeled with the necessary administrative information to support the policy, which might include information sensitivity or integrity levels, an assigned data type, or the index or name of a compartment. They also require a broad set of enforcement points across a majority of operating system operations.

An initial experimental implementation has provided the desired functionality of enforcing three fixed MAC policies: MLS, a fixed-label Biba policy, and a generalization of the native FreeBSD Jail compartmentalization policy. In the long term, we hope to provide a more general framework for introducing mandatory access control mechanism. The policies are enforced over a fairly wide set of system objects, including processes as the target for inter-process operations, system management objects such as `sysctl` nodes, file system objects such as files and directories, and network objects such as sockets, interfaces, and `mbufs`. MAC labels are described by a `struct mac` which is appropriate for use on both subjects and objects, and currently contains three fields relevant to the three policies.

To support the labeling of subjects (processes), the `ucred` structure is extended to include an additional `struct mac`. `cred0`, the process credential for the first kernel process, is initialized to high integrity, low secrecy, and is not present in any `jail` compartment. All other processes inherit this credential, unless an intermediate process has modified it; privileged processes are permitted to update the MAC fields in accordance with the MAC policies. The user login mechanisms have been updated to retrieve per-user label information from the `login.conf` user class data. This requires that components of the system making use of the `setusercontext()` call now also set the `SET_MACLABEL` flag. Eventually, additional sources of information, such as incoming terminal and network label, may be used to make a policy-driven label determination.

For inter-process authorization, the existing `p_can*()` primitives were modified to call the `mac_ucan*()` versions of the call which could return a new failure mode.

To handle the labeling of transient kernel objects, a new label structure was created, `struct objlabel`, which contains the necessary ownership and protection information, including owner and ACL, as well as a `struct mac` for mandatory protection. `struct objlabel` behaves in a similar manner to `struct ucred`, in that a set of initial object labels are initialized by appropriate kernel subsystems, and then inherited (copy-on-write) by various children objects. For example, packets inherit the object label of the interface they originate from. for objects created by subjects, the new object label is based on a composition of the subject credential, and possible object parents. A series of new access control check primitives were introduced that check authorization between subject credentials and object labels, and were liberally scattered through system operations.

Some objects, such as sockets, play the interesting role of both subject and object: FreeBSD caches the subject's credential with the socket on creation, which allows the properties of the socket to remain static when transfered or inherited; this also allows UID-based decisions to be made on delivery of packets to sockets in the `ipfw` firewall code. This permits MAC delivery decisions to be made at the network layer without directly inspecting the receiving process or dealing with the ambiguity of multiple processes having access to a single socket. However, sockets are also objects when written to or read from by processes that have access to them, and therefore have an object label. Both types of events (acting on the socket as a subject and as an object) require mediation.

Currently, file system objects do not make use of the object label abstraction, instead mapping MAC labels into EAs on the file system, reading them when an access control check must be made. A new access control primitive, `vaccess_mac()` accepts subject credentials, vnode properties, and MAC labels loaded from EAs, and returns an access control decision which is then composed with the results of the discretionary access control check, `vaccess_acl_posix1e()` to generate a final access control result. In the future, we will look at allowing file systems to maintain `objlabel` structures directly, improving their ability to utilize more general abstractions.

Many MAC implementations make use of poly-instantiation to resolve namespace use conflicts by processes with conflicting labels. For example, UNIX processes may expect to be able to write files to `/tmp` at will—however, information flow policies may not permit a process with one integrity level to be aware of files written to the directory by a process with a lower integrity level. If the two processes select the same file name, under traditional UNIX semantics, one process will receive an error: this is not permitted under information flow MAC policies. Poly-instantiation allows different processes to appear to address the same namespace while being partitioned from one another: in the case of the file system, this might mean that the `namei()` name lookup routine points the processes at different underlying directories. TrustedBSD does not currently implement automatic poly-instantiation for directories, or for other namespaces such as the IP port and System V IPC namespaces, and in that sense, is incomplete. For the purposes of processes making use of the `/tmp` directory, appropriate setting of the `TMPDIR` environment variable has proven sufficient for the present—however, in the future, this issue will need to be addressed.

Since this is still a highly experimental environment, performance figures are not yet available, but appear to be similar to those of ACLs: when not involving file system accesses, the performance cost for most objects is negligible; when an EA operation is required, the performance corresponds to the required EA operations. The impact on the network subsystem is of particular interest, as new label operations are now interposed on existing packet and interface operations, and may impose a performance hit. Future MAC work on FreeBSD will include improved abstractions for managing labels, more pervasive use of these abstractions, such as in the file systems, and implementation of features such as poly-instantiation, processes making use of ranges of labels to mediate access between normally isolated process classes, and work to measure and optimize performance.

5 Lessons Learned

A number of important lessons relating to both open source and security development can be derived from the experience of introducing the current TrustedBSD feature set into the FreeBSD operating system, and will be carried into future work.

- *Adapt to an ill-defined starting point.* While FreeBSD is characterized by strong architectural design, security functionality has never been a specific target. As a result, substantial cleanup was required to bring the starting point in line with expectations. This is actually a benefit, as it provides the opportunity to educate the broader community about security requirements, and to grow a more complete understanding of current use.
- *Incremental improvement can result in good software.* While incremental development can introduce security problems, as seen with inconsistent access control checks, introducing interfaces and features incrementally during the overall software development process of a larger project reduces the workload for developers by reducing the cost of testing and source code merging. That is to say, "release early and often" can be rephrased as "release early and merge often" for

projects that build off an existing project that constitute a moving target.

- *Improved abstractions for existing features support most new features.* Current security functionality in FreeBSD provided rationale for the abstraction improvements necessary for new security work. Making these abstraction improvements has had a high payoff in terms of improving flexibility to introduce new security models and services.

- *Being part of the developer community provides credibility to make more far-reaching changes from within, rather than "throwing the changes over the fence".* To have greatest impact, it is necessary to be part of the development community, working with that community to reach design decisions that are acceptable to all relevant parties.

- *Cross-platform portability efforts have high pay-off.* Working to build cross-platform consensus on security interfaces can have a high payoff: this has already proven the case with Samba support for POSIX.1e ACLs, and is likely to continue to be the case moving forward with security integration into other common applications.

- *Time invested in public relations activity is time well spent.* The open source community is driven by a desire for features and improvement, and revolves around a relatively small set of online forums. Investing time in publicizing work and building credibility can have very positive results in terms of generating broad support for the features, and may be vital when it comes to portability work.

6 Future Directions

TrustedBSD remains very much a work in progress: while a number of features and interfaces have been successfully integrated, much of the work remains to be done. There are several areas in which particular attention will be focused:

- *Extended attribute interface.* Substantial progress has been made in defining and implementing a general meta-data service for UNIX-style file systems. However, substantial portability work remains to be done so that applications can be assured consistent interfaces for accessing meta-data on FreeBSD and other platforms. For the TrustedBSD project, this will involve porting the EA mechanism to other operating systems, including the currently targeted OpenBSD and Darwin platforms; it will also involve continued discussion with members of the Linux community.

- *Mandatory access control.* With one and a half experimental MAC implementations under the bridge, the experience gained is becoming sufficient to look at integrating some of the components of the MAC implementations back into the base system. In particular, this involves abstraction improvements such as generalized object labeling, which permit the association of security labels with arbitrary kernel objects. MAC is an important feature to many potential TrustedBSD consumers, and therefore represents the next major integration challenge.

- *Audit.* An important trusted operating system feature unmentioned in this paper is that of event auditing. While implementing event auditing has been a goal of the project since it started, audit will most likely represent a serious challenge. In part, this is because auditing support requires widespread and intrusive changes throughout the kernel to gather information, as well as posing a substantial performance burden.

- *Plugability.* A long-term goal of the project is to improve the modularity of security services within the kernel so that they may be easily extended or replaced. This is possible through improved abstractions, and many examples of extensible kernel subsystems exist on which to base this work, including the Virtual File System (VFS)[21].

- *Documentation.* As the availability of TrustedBSD features increases for the broader FreeBSD community, documentation and education will play an increasing role in the project's work, to keep both system developers and users abreast of the new services.

7 Related Work

There is a long history of research and development relating to trusted operating systems. In the area of access control, there has been extensive research into various types of discretionary and mandatory control models[3][4][6], evolution of these models into standards and requirements[16] [9][5][11][10], and improved abstractions and models derived from these experience[2][13].

A number of trusted systems have been developed in the form of both research operating systems, and extensions to existing commercial systems. Trusted Mach[19] and other experimental trusted operating systems have explored the impact of secure design when building from the ground up. Many UNIX vendors offer trusted versions of their systems built in-house, such as SGI's Trusted IRIX[18]. There are also operating system security extension products that introduce trusted operating system features, such as PitBull from Argus Systems[1]. Open source trusted system work includes the LOMAC extensions for Linux[6], SELinux[13], POSIX.1e ACLs[8] and Capabilities[15] for Linux, and the Linux RSBAC project[17].

The TrustedBSD project benefits from both past and current research, building on the exploration of access control requirements and models, as well as research into improved abstractions and interfaces.

8 Conclusion

TrustedBSD provides a set of trusted operating system extensions to the FreeBSD operating systems. Through close cooperation with the FreeBSD community, a tight integration between the security features and base services of the operating system will be possible. The challenges in such an environment are both technical and social, where tasks from both categories play an important role in the success of the project. Not least of the challenges is the education of FreeBSD developers and users regarding new features.

9 Acknowledgments

Large parts of this work were done in cooperation with the FreeBSD and TrustedBSD development communities. In particular, I thanks Chris Faulhaber, Thomas Moestl, Ilmar Habibulin, and Brian Feldman for their participation in developing and debugging these features. In addition, my thanks to Ruslan Ermilov, Dima Dorfman, and Chris Costello for documentation support.

A major focus of the TrustedBSD work has been to emphasize portability of the feature sets, particularly with other open source operating systems. Both Andreas Gruenbacher, author of the Linux ACL and EA implementations, and Andrew Morgan, author of the Linux Privileges implementation, have been vital to this approach through their discussion of the POSIX.1e specification, and implementation feedback and critique. Thanks also to the Trust Technology group at SGI, including Casey Schaufler, Richard Offer, and Linda Walsh, all of whom have provided feedback on the POSIX.1e specification, and system/application requirements.

Substantial contributions of funding, development resources, and travel and communication reimbursement have been provided by NAI Labs, BSDi, Safeport Network Services, without whom the TrustedBSD Project would not have been possible.

10 Availability

The TrustedBSD Extensions to FreeBSD are distributed under a two-clause Berkeley-style license, encouraging integration into both open and closed souce products. During development, patches and code are available via the TrustedBSD web site: `http://www.TrustedBSD.org/`

As features reach maturity, they are integrated back into the base FreeBSD distribution: `http://www.FreeBSD.org/`

References

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

[2] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement UNIX prototype. In *Computing Systems, Winter, 1996.*, volume 9, Berkeley, CA, USA, Winter 1996. USENIX.

[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 considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, Apr. 1977.

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

[6] T. Fraser. LOMAC: MAC You Can Live With. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.

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

[8] A. Grunbacher. Extended attributes and access control lists for linux. http://acl.betbits.at/.

[9] IEEE. Portable operating system interface (POSIX)–part 1: System application program interface (API): Protection, audit and control interfaces: C language, October 1997. PSSG/D17, POSIX.1e.

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

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

[12] P.-H. Kemp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings, SANE 2000 Conference*. NLUUG, 2000.

[13] 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.

[14] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. Design and implementation of the 4.4BSD operating system, 1996.

[15] A. Morgan. Privileges for linux. http://www.kernel.org/pub/linux/libs/security/linux-privs/.

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

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

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

[19] Trusted Mach Security Architecture. TIS TMACH Edoc-0001-97A.

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

[21] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In *Proceedings: USENIX Association Winter Conference, January 23–25, 1985, Dallas, Texas, USA*, pages 117–124. USENIX, Winter 1985.

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