LAFS: A Logging and Auditing File System

Christopher Wee

Department of Computer Science
University of California, Davis
wee@cs.ucdavis.edu

Abstract

The Logging and Auditing File System (LAFS) extends the familiar file system with policy-directed security logging and audit analysis. Our primary motivation is ease-of-use and effective auditing services for non-privileged users. The LAFS file system supports regular Unix-style file semantics, assists the user by configuring file system protection (permissions) according to a user-specified policy, logs file and directory accesses, and audits the logs against the same policy. Although initial installation requires privileged access, LAFS is a novel approach to auditing wherein day-to-day configuration and usage is performed by non-privileged users. We also describe a policy language based on predicate logic and regular expressions that is suitable for expressing Clark-Wilson style integrity policies. From the user's standpoint, LAFS works in a heterogenous environment of different hosts and operating systems. It is portable to any host that supports the Network File System (NFS) interface. Audit management is simple as LAFS uses local or remote file servers for storing encapsulated audited objects — a combination of file data and access logs. Archival and restores of audited files work in a normal and expected manner, so that audit logs and audited data remain mutually consistent. Prototype implementations of LAFS for SunOS 4.1.3, Solaris 2.4 and SGI IRIX operating systems have been built. Finally, we discuss theoretic problems with the security of the LAFS system itself.

1. Introduction

The Logging and Auditing File System (LAFS)[1] is a file system with an embedded security auditor. We recognize that users may have security needs that are different from those in the system security policy. Given a user-specified policy, LAFS helps the user configure file system protection mechanisms (i.e., file permissions in Unix), performs logging of file accesses and audits the file access logs against the security policy upon request. Our aim is to make security auditing simple, convenient, and effective. Logging and audit analysis are completely transparent to the user. Since users are already familiar with the file system, LAFS audit logs and audit reports appear as (virtual) files in the file system. Auditing in LAFS is limited to the file system unlike a system-level auditor that monitors all system objects. This is not a serious disadvantage as many users are content to audit just file usage. Extensions to other objects such as network channels are under consideration.

In section 2, we describe a scenario that motivates user-policy discretionary auditing, and in section 3, we describe the embedded auditor and the benefits of our approach. Our policy language is described in section 4 and an analysis of the trust and security is presented in section 5. We describe our prototype implementations, in section 6, and present our conclusions and suggestions for future research in section 7.

2. A scenario

A team of programmers wish to audit the use of their master development code tree. They want to detect accesses to the development tree that dilute its integrity, e.g., using an editor to make changes in revision controlled files, use of un-sanctioned compilers. The team works in a distributed, heterogenous environment and the development tree is distributed across several networked file servers, without any commonality in the system-level auditing facilities available on each machine or server.

Furthermore, the team realizes that the audit logs permit the evolution of each module to be traced with accountability for each incremental change. Revision control tools such as SCCS and RCS do trace such evolution, but only for source code files. Similar tools for compiled object files, directory structure, and programming tools (compilers, linkers) are not available.

Finally, they cannot tolerate any performance degradation for activities unrelated to the master development tree, i.e., team members who are editing, compiling or debugging in their private build directories should not need to be logged or

1. This work has been supported by the University Research Program of the National Security Agency as part of the Audit Workbench project contract # DOD-MDA904-92-C-5148.

audited.

Ease-of-use, auditing that fits inconspicuously into their existing development environment without operating system kernel modifications, applications modifications and support for existing file servers and archival procedures are desirable.

Summarizing our programming teams requirements:

- User specified security policy for their files. Other programming teams may have different policies.
- A single policy controls logging and generates oracles during security audits.
- The auditor must be easy to use.
- Works in a heterogenous and distributed environment, audit logs and reports accessible from any host.
- Minimal performance degradation, especially for unaudited files.
- Transparent to unmodified applications.

3. User-directed auditing

In this section, we discuss the difficulties with system level auditing and auditing in a distributed environment as motivated by our group of programmers in the previous scenario. We describe the type of auditing (policy and queries) that users expect.

3.1 Users cannot use system-level auditing

System level auditing (SLA) is not suitable for non-privileged users, such as our programming team, because it requires administrator privileges to install, configure, and use. We have not found any SLA auditor that permit groups of users to configure discretionary audit policies for themselves without assistance from the system-administrator.

SLA, which is appropriately designed for a single system or network-wide mandatory security policy, does not accommodate multiple discretionary audit policies for groups of users working in many roles. Typically, SLA logs are collected into a single log that requires special audit tools to read and manipulate. The audit log becomes a new and distinct resource in the operating system, with which ordinary users are unfamiliar. Raw audit logs and reports cannot be disseminated to interested parties as may contain private information belonging to other users [9].

SLA can be configured differently for each user (e.g., complete logging of the privileged users and cursory auditing of ordinary users). However, it is not possible to tailor logging for different activities. For example, in a typical C2-level auditor such as SunSHIELD [19], event-logging cannot be predicated on user-activity (e.g., complete logging for code development and no logging during Web surfing). LAFS auditing approximates activity-based auditing by recognizing that users, in different activities, use different sets of files.

3.2 Heterogenous & distributed auditing

A secondary objective of LAFS is consistent auditing as users work across networked hosts and different operating systems, performing distributed operations that manipulate both local and remote objects. Two significant hurdles in this environment are audit log aggregation and the interpretation of the log [17].

Distributed audit log aggregation is usually solved by transmitting logs to a central aggregation site[18]. Centralized audit collection introduces the problem of dissemination of audit results to the distributed users. In our experience, most users prefer to work on their local workstation than on a central server. Thus the user may wish to aggregate audit logs of their work from several distributed hosts at one host, but each user may select a different host as his aggregation site! The analysis at the log collection point is also complicated by the requirement to understand the semantics of all the logs sent from a wide variety of hosts and operating systems.

Combining, analyzing and interpreting logs from different hosts is difficult because of the diverse types of resources (files, users, devices, services) and operations (system calls, system libraries, applications) present in a heterogenous environment. LAFS exploits the relative similarity of file system designs and audits the ubiquitous NFS protocol [21] that is popular in most heterogenous environments. The NFS protocol describes a generic file system model with operations such as read, write, create, remove etc. Consequently LAFS audit logs use the same operations and semantics.

3.3 Policies and audit queries

A security policy describes whether users are authorized to access files and resources. In LAFS, a security policy is a set of predicates (rules) that authorize or prohibit transactions by users and programs upon files within a directory hierarchy. The policy rules are similar to Clark-Wilson (C-W) access triples [5], with some minor extensions. Rules that form each user's *policy* dictate what transactions should be logged in order to verify that the security policy was correctly enforced by the OS protection mechanisms. The user policy also state what kinds of accesses are authorized, independently of the underlying file protection configuration. These user policy rules are verified during audits. The user policy may also specify additional goals and objectives (e.g., "tabulate accesses to the FTP server files that occur during business hours"). Thus, a LAFS policy rule expresses a desired security constraint, as well as establish logging and analysis requirements.

LAFS makes it possible for any group of users to implement policy-directed auditing of files. Each directory hierarchy in LAFS has a separate discretionary policy specified by the owner of the file hierarchy. Each policy in each sub-tree does not conflict with other policies in other sub-trees as they cover their own files.

```
#action:date:filename:user:program:op
allow:*:[Mm]akefile:*:make:read
allow:*:[Mm]akefile:Owner:makedepend:write
allow:*:*.o:Group:gcc,ld:write
allow:-960630:*.c:Owner:emacs,vi:write
allow:-960701:RCS/:Lib:rcs,co,ci:write
```

Figure 1. Policy rules

Figure 1 shows some examples of policy rules in our policy language. The language is described in detail in a later section. The first rule permits any user running the `make` program to read files named `Makefile` or `makefile`. The last rule allows the team librarian to write to files in the `RCS/` directory after July 1, 1996 (presumably a source-code freeze date) using the revision control tools.

When the access logs are reviewed against the security policy, violations are collected into the audit report. Alternatively, the user may directly query the audit logs to determine which user last modified a file, or list all files a user has read, or discover which user created a file (regardless of current file ownership) or any other fact regarding the past history of file accesses.

4. Logical organization of LAFS

The LAFS auditor is stacked on top of an existing file system (FS) as shown in Figure 3. To the application program, the LAFS file system appears to have an extended interface, i.e., in addition to FS services, LAFS auditing services. Since LAFS appears to be just an ordinary FS client to the underlying FS, no kernel modifications are necessary. Stacking file systems permits extensibility while maintaining backwards compatibility with existing applications.

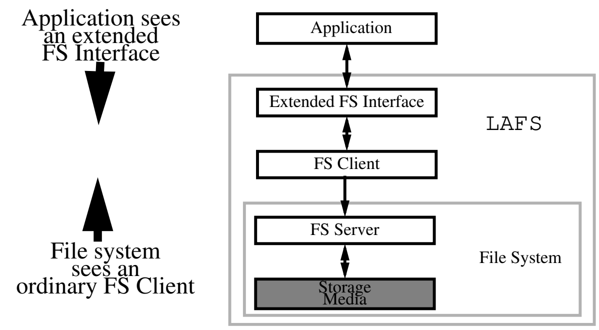

Figure 3. LAFS stacked onto File System

n Figure 2, we examine the stacking in detail. The underlying file system (bottom of figure) consists of a naming service that maps file-names to file-objects and a file manager that performs the actual reading and writing of data. The naming service is also where access control is enforced. Stacked on top of that file system, is the LAFS interface consisting of a naming service, a file manager, and a small interface to attach and detach directories (used by `lattach` and `ldetach`) To permit transparent operation of existing

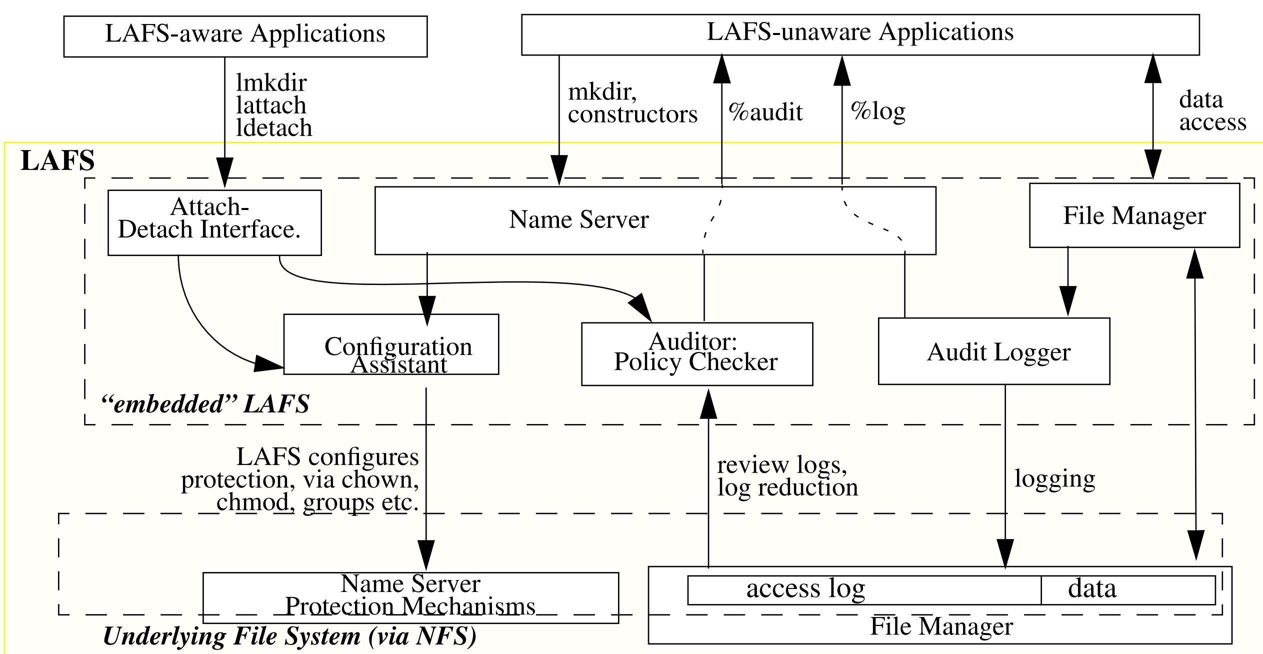

Figure 2. LAFS file system stacked onto existing OS file system.

applications, the rest of LAFS interface is backwards-compatible with the existing FS (i.e., no augmented `open()`, or `ioctl()` functions). To avoid extending the file interface with added functionality for logging and auditing, we extend the naming service to access the new functions with the addition of special suffixes shown in Table 1.

Table 1: Special filename suffixes

Suffix	Meaning
-no suffix-	access file
%log	access historical log
%policy	access security policy
%audit	perform audit and supply audit report

For example, appending the suffixes "%log" to a filename and performing a regular read operation, the user accesses the historical access log for that file. We will describe these suffixes shortly. The mechanism is quite general and we use extended naming to initiate almost any arbitrary function through the FS interface. See "The user's view" in Section 4.1.

The file manager in embedded LAFS passes all requests to the underlying FS without modification. File accesses are logged but not mediated by LAFS. We rely on the protection mechanisms of the underlying file system, e.g., Unix permissions. When the user requests an audit, the auditor reviews the access logs against the security policy and collects violations into an audit report. The auditor may also summarize logs on user request.

4.1 The user's view

LAFS audits files contained within a single directory subtree as shown in Figure 4. To begin auditing, the user prepares

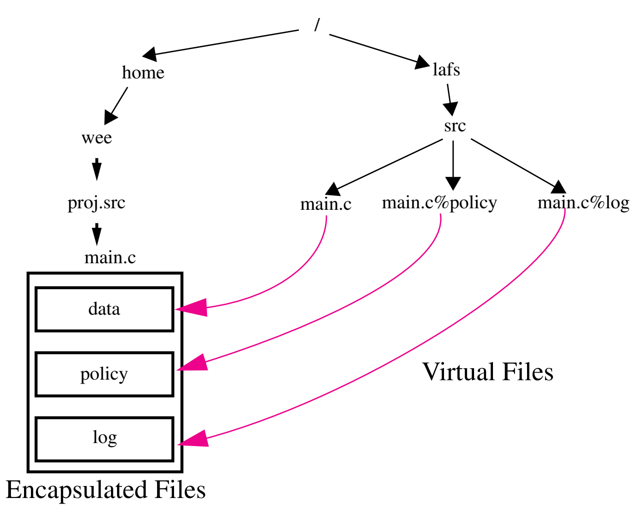

Figure 4. Encapsulated and Virtual files.

a directory to store the files and access logs, and then attaches it to LAFS.

```
lmkdir proj.src policy
```

The directory `/home/wee/proj.src` is created and the security policy is stored in that in a hidden file. To use the directory, it must be attached to LAFS using the lattach command:

```
lattach /home/wee/proj.src /lafs/src
```

A virtual copy of the directory appears at the path `/lafs/src` (see Figure 4). Files in `/home/wee/proj.src` are reflected as virtual copies in /lafs/src, except that accesses to the virtual copies are audited. When auditing must be suspended, the directory is detached from the auditor using the `ldetach` command:

```
ldetach /lafs/src
```

Subsequent sessions are prefaced with a `lattach` command and terminated with a `ldetach`.

Each file stored in `/home/wee/proj.src` is an encapsulated file. Each file has three components — the file data, the historical access log for that file, and its relevant security policy rules, as shown in Figure 4. These parts of the file are accessible to the user by appending a special suffix to the file name (a list of suffixes is shown in Table 1). Virtual filenames do not appear in directory listings.

For portability and ease-of-use, the virtual files are plaintext with a simple one record per line format. The access log of a file named "Makefile" is shown in Figure 5. Each record

```
May 4 15:01:37 1995:wee:Configure:create:ok
May 4 15:01:37 1995:wee:Configure:write:ok
May 4 15:01:37 1995:wee:makedepend:write:ok
May 4 15:01:38 1995:wee:vi:read:ok
May 4 15:01:39 1995:wee:make:read:ok
May 4 15:01:40 1995:wee:make:read:ok
May 4 15:01:41 1995:root:make:read:ok
May 4 15:01:42 1995:wee:textedit:read:ok
```

Figure 5. Access log for Makefile, i.e., Makefile%log

represents a file access, with the date and time, the accountable user and program, the operation performed (NFS operations) and whether the operation succeeded. The simple audit record structure permits further analysis using Unix tools like awk, sed, or perl.

To perform an security audit, the user accesses a special file with the suffix "%audit". For example, to audit all files in a sub-tree, we access the file ".%audit" in the top-level directory:

```
cat /lafs/src/.%audit
```

The virtual file ".%audit" performs a security audit and contains the audit report. If no accesses occurred since the last audit, the audit is skipped and the previous (cached) audit report is returned.

```
LAFS Audit Report: May 5 21:39:13 1995
---------------------------------------
1 May 4 14:58:44 1995:wee:gcc:read:ok
2 May 4 14:58:45 1995:wee:ld:read:ok
3 May 4 14:58:47 1995:wee:gcc:write:ok
4 May 4 15:01:38 1995:wee:vi:read:ok
5 May 4 15:01:42 1995:wee:textedit:read:ok
5 violations / 49 log entries
```

Figure 6. Audit report

Each user attaches his own copy of the audited directory. We are currently experimenting with a customizable user's view of the shared directory, so that the permissions confine the user's behavior. Virtual files may have different ownership and permission attributes than the underlying encapsulated file. For example, if security policy forbids writing to a particular file, the virtual copies will not permit writing but the actual encapsulated file will permit writing by the owner so that logs can be updated. Virtual file ownership and permission attributes is more flexible than using Unix-style user groups and file group permissions.

4.2 Auditing versus enforcement

We refrain from using the policy rules to augment the underlying protection mechanisms, e.g., permissions, access-control lists (ACL), user and group ownership. We intend LAFS to be a privileged, but benign system component outside the trusted computing base. Although LAFS is a fairly simple file system, it is not immune to implementation errors and we feel that additional complexity to augment the protection mechanisms and enforce policy may introduce additional vulnerabilities. Some versions of UNIX, e.g., HPUX, have extended the Unix owner/group/world protection model with extended ACLs, which LAFS will interpret and use.

Despite our decision not to enforce policy, LAFS assists the user to configure the OS protection mechanisms. In our Unix prototype, we have discovered that the Unix ACL is not sufficiently flexible to adequately enforce the security policy precisely[2]. Consequently, with respect to the security policy, unauthorized activities are permitted by the protection configuration. A more flexible protection mechanism, perhaps direct support for access-triples in Tunis [22], would permit LAFS to configure the protection mechanisms precisely. Alternatively, we can use *phantom users* and group permissions to simulate C-W access triples using the existing Unix permissions [14]. However, the inflexibility of the Unix permissions permits us to detect violations during our audits!

2. A system is secure if all reachable states are authorized. A system is precise if all authorized states are reachable [6].

If the security policy changes after logging has begun, we must take care not to invalidate prior audits — previous transactions may now violate the new policy, or previous violations may become permissible. Policy changes are effective from the time of alteration (until the next change) so that prior audit reports remain valid. Retroactive policy changes are not permitted. This is accomplished by using the date field of the policy language. In Figure 7, the first and fourth rules cover conducted prior to January 1, 1995 and the second through fourth rules covers activities after January 1, 1995. Rules that do not change are not qualified by date.

```
allow:-950101:myfile:Owner:app1:read,write
allow:950101-:myfile:Owner:app1:read
allow:950101-:myfile:Owner:app2:write
prohibit:*:secrets:!Owner:*:*
```

Figure 7. Combination of policy-1 and policy-2

Alternatively, the user can explicitly change the policy by creating another LAFS directory hierarchy and copying all the data to the new directory covered by a new policy.

5. The policy language

This language is used to specify authorized transactions and prohibit undesirable behavior. LAFS uses the policy in three ways — to generate oracles during an audit, to configure logging, and to configure FS protection mechanisms. The language is based on predicate logic and regular expressions.The abstract model of policy we use is similar to Clark-Wilson access triples. Recall that a C-W access triples consist of a subject, a transaction program and a set of objects, i.e., (S, TP, (CDI_1, CDI_2,...)).

A *policy* in our language is a set of predicates, each predicate having six attribute expressions — date, filename, username, program, operation and status. The syntax of the policy language is shown in Table 2:

Each attribute expressions may be specified using extend-

Table 2: Policy language syntax

```
Policy:  Rule | Rule Policy
Rule:    Action:Date:File:User:Prog:Op:Stat
Action:  allow | prohibit | count | warn
Date:    * | data
File:    File, filename | filename | Dir | *
User:    Name | User, Name | *
Name:    uid | name | uid@host | name@host |
         Owner | Group
Prog:    program | File, program | *
Op:      NFSOp | Op, NFSOp | *
NFSOp:   read | write | create | remove |
         link | readlink | lookup | symlink
         | mkdir | rmdir | readdir | rename
Stat:    ok | error-msg | *
```

ed regular expressions[3]. The values of the attributes are instantiated in the audit logs and if all the attributes match, then the action specified in the action attribute is performed. Currently, we support only four transaction actions — allow, prohibit warn and count. If prohibited or warned transactions are discovered in the audit log, the transactions are reported in the audit report. Counted transactions are also summarized in the audit report.

The language has several built-in predicates such as *Owner, Group and Directory*. The *Owner/Group* predicates are true when the file's owner or group attributes match the user attribute of a policy rule. The *Directory* predicate, represented by a trailing slash on a filename or pattern, is true when the instance of the file attribute is a directory.

Our policy language is purposely simple and sacrifices expressive power for ease-of-use. However, we did add three more attributes to C-W for convenience. The date attribute permits date and time ranges to be specified, e.g., June 1-30, 1995, or 9am-6pm. Rules may be conditional on different parts of the day such as:

```
prohibit:0900-1200,1300-1700:*.wad:*:doom,
```

which reads as "prohibit *Doom* game-playing except outside of business hours or during lunch."

The addition of the "op" tuple addresses a serious shortcoming of using Unix applications as TPs. The C-W model assumes that TPs embody specific, fine-grained behavior, and access triples allowing or prohibiting TP and CDI combinations ensure integrity. However, in reality, applications are monstrous beasts, each capable of performing many functions. For example, without the "op" attribute, what does the following policy rule mean?

```
allow:*:foo.c:*:emacs
```

A user can use *emacs* to emulate virtually any other application on the system since emacs itself has a built-in general purpose interpreter. Since we cannot reasonably expect existing applications to be reduced into smaller functional units, we specify application transactions with a finer granularity with the "op" or NFS operation attribute.

Combining "prog" and "op" attributes narrows the range of behaviors and constrains the permissible effects of the application program. For example, we would like to model a TP that alters the password file and another that just reads the password file. The tuple for the set-password rule might be,

```
allow:*:/etc/passwd:root:vi,emacs:write:ok
```

and the use-password rule,

```
allow:*:/etc/passwd:root:vi,emacs:read:ok
```

The results of transactions, such as error status or valid status, are also specified as a part of the access-tuple. By specifying command status, the user audits failed commands or attempts to violate the protection mechanisms. For example, the predicate,

```
prohibit:*: *:*:*:write:error
```

will flag failed attempts to write to any file in the audit report.

Our desire for simplicity and ease-of-use sacrifices the expressiveness of the policy language. For instance, one cannot easily specify the restriction "one can only delete files one created". We simulate the effect if we rely upon the Unix semantics that the creator of the file is also its owner, however, the creator can give the file to another user using `chown`. We are considering including another predicate *CreatedBy* as in [11].

The language does not prohibit internal inconsistency, and we would like to have a compile-time checker identify conflicting rules to the user. LAFS translates the policy rules into a checking script using a simple (approximately 40 lines) Perl script [23]. For example, the rules shown in Figure 10 are

```
#action:date:filename:user:program:op
allow:*:[Mm]akefile:*:make:read
allow:*:[Mm]akefile:Owner:makedepend:write
allow:*:*.o:Group:gcc,ld:write
allow:-960630:*.c:Owner:emacs,vi:write
allow:-960701:RCS/:Lib:rcs,co,ci:write
```

Figure 8. Policy rules

translated into policy checking Perl script shown in Figure 11. The first policy rule "allow..." is translated into an action

```
next if $Filename =~ m/^[Mm]akefile$/ && /
.*:.*:make:read/;
next if $Filename =~ m/^[Mm]akefile$/ && /
.*:$uid:make:read/;
next if $Filename =~ m/^[Mm]akefile$/ && /
.*:$uid:makedepend:write/;
next if $Filename =~ m/^.*\.o$/ && /
.*:$Group:(gcc|ld):write/;
next if $Filename =~ m/^.*\.c$/ && /
.*:$uid:(emacs|vi|textedit):write/;
```

Figure 9. Policy checking code

that skips to the next audit record if the current audit record matches the attributes specified in the rule.

We continue to refine the policy language as we gain experience with it, with the objective of making it simpler and easier for the user to use, not necessarily more expressive.

6. Prototype

The LAFS system has evolved to satisfy the following

3. Perl regular expressions [23].

goals with their attendant features and benefits:

Transparent access. LAFS behaves like a regular file system, except that logs and audit report files are created and updated automatically.

User-specified policies. The user may specify security and audit policies in the policy language (described in the next section). Separate policies may be used for separate groups of files.

Configuration assistance. Users do not require special privileges to configure logging or perform security audits in LAFS. Furthermore, LAFS assists the user in configuring directory/file permissions and ownership to enforce his security policy.

Familiar framework. The naming policy of the file system is simply extended with suffixes that trigger functions such as log review, or audits.

Distributed aggregation. Since file accesses are processed within LAFS, all auditable events that pertain to a file are automatically aggregated. No post-processing of a central audit log is required. Network accesses are logged as easily as local accesses, avoiding the distributed naming problem with other aggregation schemes, i.e., resolving different pathnames to the same file object from remote hosts is impossible in many environments due to dynamic name servers like automounter[20]. Audits are performed from any networked host. All clients have the uniform consistent view of the audited file system.

Heterogenous auditing. Auditing the generic NFS file system model and protocol lets LAFS function in a heterogenous, distributed environment consisting of different hosts and operating systems.

Minimal performance impact. Unlike kernel based auditors, only accesses to LAFS file system incur a performance penalty. There is no performance degradation while accessing any other file systems or for users who do not use LAFS. This satisfies the requirements of the programming team discussed in our earlier scenario.

Assists in damage recovery. After an intrusion, the system administrator may restore files from archives to eliminate trojan horses or trapdoors. If selected portions of the file system is restored, how are selected portions of SLA logs restored? If portions of SLA logs are not selectively restored, they may be inconsistent with respect to the file system, e.g., the log may indicate that a file was deleted, but the file has been subsequently restored.Our encapsulated files combine logs and data together so they always remain mutually consistent. Archival and restoration of the encapsulated files work in a normal and expected manner.

Portable. No special system calls or no kernel changes are required for LAFS unlike [1]. Any local or remote file system may be used for the underlying storage.

Archival. Encapsulating file data and logs is motivated by archival procedures. As a single entity, data and logs will always remain consistent with each other. Thus, when files are restored from archives, the logs are restored as well.

Exchange. If the user wants to give away a copy of a file, and continue logging accesses to the copy, he gives away a copy of the encapsulated file. The recipient must use LAFS to access the data to the copy or if LAFS is unavailable, use some utility programs to read and modify the copy. These utilities access the data portion of the encapsulated file and update the encapsulated log accordingly. When the copy is returned to the owner, the log can be accessed and audited.

6.1 Architecture

We have constructed prototypes of LAFS for Solaris 2.4, SunOS 4.1.3 and IRIX 5 operating systems. These prototypes enabled us to model and test different audit policies as well as learn from first-hand experience with discretionary, user-specified policy and auditing. The prototype is derived from Cryptographic File System sources. The reader is referred to CFS [2] for a detailed discussion of the subtleties of emulating NFS interface and the performance considerations due to file caching.

The LAFS prototype consists of a daemon process and a small collection of tools to allow the user to communicate with the daemon, configure auditing, reduce audit logs and perform auditing. The daemon process, `lafsd`, emulates a NFS file server and communicates with the Unix kernel via interprocess communication in the NFS protocol. The NFS protocol consists of 17 *remote procedure calls* (RPCs) that implement file system primitives such as read, write, create etc. The client initiates the communication and the server responds to each request in a *stateless* manner, that is, all information required to satisfy a request is included with each client RPC. An additional RPC interface allows the user to attach and detach directories from the LAFS daemon and configure policy. This technique of using the NFS protocol interface to implement new file services has been used by 3DFS [16] and CFS [2].

It is instructive to examine the dataflow between the LAFS daemon and the underlying file system and compare that to the normal dataflow between an application and the file system. Each client machine runs one copy of `lafsd`, mounted on the *localhost* interface and the `/lafs` directory, which accepts LAFS requests to mount and unmount user's audited directories. In Figure 12, arrows represent data flowing across operating-system contexts, kernel modules or networks. From the top, the application makes system calls for file services. In the case of LAFS, the FS client in the kernel relays requests to the LAFS daemon NFS server interface (`lafsd` was mounted as a NFS server at `/lafs`), and it in turn uses the local system call interface for its storage needs. From this point, lafsd appears as just another client requesting file services.

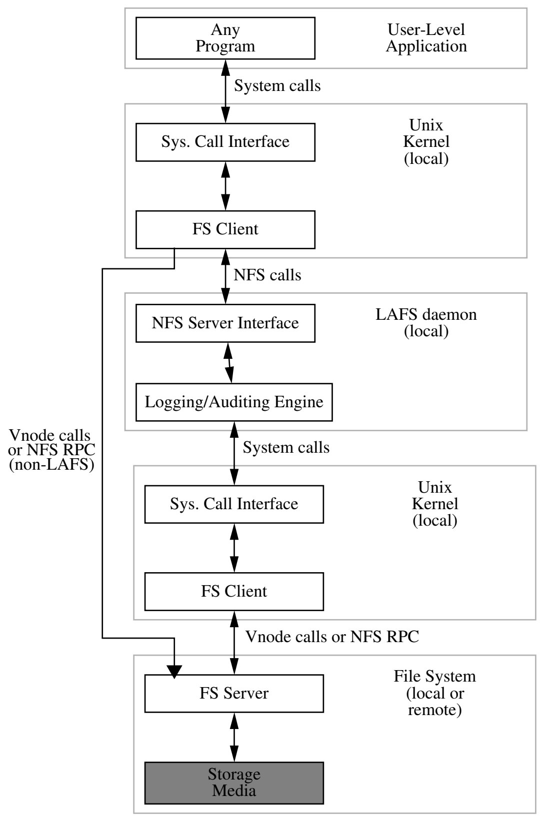

Figure 10. Dataflow in LAFS prototype

When an application is not using LAFS, calls are routed from the FS client directly to the file server, bypassing `lafsd`. Thus, users of LAFS incur two additional context switches and two more disk buffer copy operations than the normal case. No performance degradation due to auditing occurs when auditing is inactive or when using a non-LAFS daemon.

Our experience with the prototype shows a slight performance penalty especially when using a remote file server as the underlying storage medium; the network protocol dominates the transaction. Caching by the FS clients, read-ahead and write-back, also hide much of the additional delay. Not shown in the diagram are the additional audit logging that occurs when transactions are recorded. Our prototype does not perform any queuing at the moment, so almost every relevant disk transaction is accompanied by another for the log. Finally, there is a significant performance penalty when we query the OS kernel to learn which program is requested the file access (that information is not provided through the NFS interface). The current prototype must execute a separate program that interrogates the kernel upon every file access resulting in unacceptably long delays. Omitting the kernel interrogation step, our prototype LAFS file system is about a third of the speed of the underlying file system. We are investigating the use of the process (/proc) file system as an alternative to executing a separate program to interrogate the kernel.

7. Security and trust

Whereas our initial instinct was to regard the LAFS embedded auditor as part of the trusted computing base, responsible for enforcing the security policy, we realized that LAFS is more properly considered a broker of file services via network file system interface. As such, LAFS is a benign system component that has special privileges, but is not responsible for enforcing the *system security policy*. In fact, LAFS defers access control decisions to the underlying file systems.

The primary threats to the auditor is bypass of its logging mechanism or tampering with its audit logs. LAFS does not perform its own authentication, so it relies upon the operating system to pass legitimate user credentials, and may also be subject to masquerading.

7.1 Bypass and tampering threats

The user may attempt to bypass LAFS and access the encapsulated files directly. If he knows the format of the encapsulation, he has unaudited access to the data, or can tamper with the logs, compromising future audits.

We secure the encapsulated files by giving exclusive access to the owner of the security policy, i.e, the user who initiated LAFS auditing. The owner must be trusted not to alter the logs or breach the integrity by giving access to other users. Since the user's policy is discretionary, this is a reasonable compromise between no protections and the harsher schemes below.

More draconian measures are employed to reduce the bypass threat. We considered giving a phantom user (lafs) exclusive access to the encapsulated files, but this is contrary to our requirement that no special privileges are necessary to manage auditing.

We also tried dynamically shifting ownership of files to the LAFS daemon when auditing is active. However, this was time-consuming and very inconvenient if the lafsd program aborts (or is killed) prematurely and leaves all files locked. Processes with open file descriptors would also not have had their access revoked.

Finally, we are considering the use of cryptography to secure the contents of the encapsulated files. While LAFS is active, the daemon encrypts the encapsulated files under a random session key known only to itself. If the daemon takes precautions not to reveal the keys (e.g., preventing swapping sensitive data to a paging device), the encapsulated data is safe from unauthorized access. An untimely demise of the daemon results in a denial of service attack upon all the users of the audit, however. When auditing is halted, the files are either decrypted and held in possession of a phantom user or the key is written to a location secure from the audited users. Tampering with the encapsulated files is detectable by virtue of the encryption. The possibility of denial-of-service by the accidental loss or malicious deletion of the random key dictates the need for a emergency "back-door" like a key escrow [3].

Meanwhile, other means (e.g., system level auditing) may be used to detect tampering of LAFS logs by the owner. LAFS is also vulnerable if the security of the underlying file system is weak. Remote file systems, like NFS, are vulnerable to eavesdropping when run over an un-encrypted network, and also vulnerable to a variety of attacks due to its poor client authentication [24].

7.2 Masquerading

LAFS relies upon proper identification and authentication of its users by the operating system and a secure path between the user and the LAFS daemon. Masquerading by users will contaminate audit logs and invalidate future audit results. As with any file system, a masquerader has unauthorized access to private data. LAFS relies upon user-credentials sent through the NFS protocol. It is well known that NFS user credentials are easily forged, the daemon only accepts connections from a privileged port of the local host operating system, and rejects attempts by remote hosts to connect to it.

7.3 Denial of service

Our prototype is vulnerable to denial of service attacks because it is currently single threaded (i.e., the prototype can only service one user at a time). For example, a user may request an audit of a remotely mounted file system that he knows is unavailable. Since the LAFS system cannot proceed (the remote file system does not respond), other users are denied LAFS services. However, this vulnerability stems from our implementation and not our approach. A multi-threaded prototype with reasonable time-outs should prevent this sort of denial of service attack.

8. Conclusions

Users can express a range of security and audit policies in the LAFS policy language. Discretionary, user-specified policies are effectively enforced in LAFS. Using the policy as a guide from the onset leads to more efficient logging than the typical log, then with an audit reduction approach. However, specification of a policy is not immediately obvious to the novice user and several example policy templates should be provided.

Discretionary, policy directed auditing is very powerful and simple to use when the auditor is embedded in the file system. The embedded LAFS auditor provides simple mechanisms for logging file accesses as well as specifying and auditing user policies. File system transparency permits auditing of unmodified applications. Stacking an auditor upon an existing file system leverages the user's familiarity with file management and brings audit services to security neophytes. Extending the naming policy eliminates the need for special file system interfaces and is backward compatible with existing software. Encapsulating logs and file data naturally correlates and aggregates audit logs in a distributed environment.

8.1 Future directions

We plan to investigate many extensions to LAFS such as incrementally auditing and caching the results of previous audits to improve performance, auditing multiple policies within a single sub-tree. We are considering integrating other auditors such as Tripwire [10], COPS [8] and Hobgoblin [12] into the LAFS framework so that the user has a single consistent interface to many audit functions. For example, the "%md5" or "%cops" suffixes can trigger a Tripwire integrity check or COPS audit for that directory.

Since the LAFS user is not a trusted security officer, but regular users, we must address potential channels of information leakage between users through LAFS mechanisms. Finally, auditing network usage (e.g., WWW access, distributed objects) by embedding auditors into information servers and object brokers appears promising.

For example, combining LAFS with a HTTP server, we are attempting to log "hits" on WWW pages and keep encapsulated records of page content as well as access information. We envision this application of LAFS as a novel method gauging interest in WWW pages as the information content evolves. Both "hits" and page updates are keep in the same log.

Acknowledgments. The author would like to express his thanks to Matt Blaze for his contribution of Cryptographic File System code upon which the LAFS prototype is derived. We also thank the anonymous referees for their helpful suggestions.

Availability. The LAFS research prototype is available. Interested parties may contact the author or check *http://seclab.cs.ucdavis.edu/awb* for updates.

9. References

[1] Bershad, B.N., and Pinkerton, C.B., "Watchdogs — Extending the UNIX File System," *Proc. USENIX Technical Conference*, Winter, 1988.

[2] Blaze, Matt, "A Cryptographic File System for Unix," *Proc. First ACM Conference on Communications and Computing Security*, November, 1993.

[3] Blaze, Matt, "Key Management in an Encrypting File System," *Proc. USENIX Technical Conf.,* June, 1994.

[4] Clark, David and David Wilson, "A Comparison of Commercial and Military Computer Security Policies," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, April 27-29, 1987.

[5] Clark, David and David Wilson, "Evolution of a Model for Computer Integrity," in *Report of the Invitational Workshop on Data Integrity*, September 1989.

[6] Denning, D., *Cryptograph and Data Security*, Addison-Wesley Publishing, 1983, pp. 279.

[7] Dowell, Cheri and Paul Ramstedt, "The Computerwatch Data Reduction Tool," *Proceedings of the 13th National Computer Security Conferenc*e, October 1990.

[8] Farmer, Dan, and Spafford, E. "The COPS security checker system," *Proceedings of the Summer USENIX Conference*, pp. 165-190, Berkeley, CA 1990.

[9] Fisch, E.A., G.B. White, U.W. Pooch, "The Design of an Audit Trail Analysis Tool," *10th Annual Computer Security Applications Conference*, Orlando, Florida, December 5-9, 1994.

[10] Kim, Gene, and Spafford, E. "Experiences with Tripwire: Using integrity checkers for intrusion detection." *Systems Administration, Networking and Security Conference III, Usenix*, April 1994.

[11] Ko, C., G. Fink, K.Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring," *10th Annual Computer Security Applications Conference*, Orlando, Florida, December 5-9, 1994.

[12] Leadly, Scott, Rich, K., & Sirota, M., "Hobgoblin: A File and Directory Auditor." University Computing Center, University of Rochester, 1991.

[13] NCSC, *DOD Trusted Computer System Evaluation Criteria*, Technical Report DoD 5200.28-STD, National Computer Security Center, December 1985.

[14] Polk, T., "Approximating Clark-Wilson Access Triples with Basic Unix Controls," *Proc. Usenix Security Symposium IV*, Santa Clara, California, October 4-6, 1993.

[15] Proctor, P., "Audit Reduction and Misuse Detection in Heterogeneous Environments: Framework and Application," *10th Annual Computer Security Applications Conference*, Orlando, Florida, December 5-9, 1994.

[16] Roome, W.D., "3DFS™: A Time-Oriented File System".

[17] Schaen, "Network Auditing: Issues and Recommendations," *Proceedings of the 7th Computer Security Applications Conference*, San Antonio, TX, December 1991.

[18] Sibert, W. Olin, "Auditing in a Distributed System: SunOS MLS Audit Trails," *Proceedings of the 14th National Computer Security Conference*, Baltimore, Maryland, 1988.

[19] Sun Microsystems Inc., *SunSHIELD: Basic Security Module Guide, Version 2.4,* Mountain View, California, 1994.

[20] Sun Microsystems Inc., *Automount File System, Answerbooks 4.1.2 & 2.4*.

[21] Sanberg, R., D.Goldberg, S.Kleinman, D.Walsh, B.Lyon, "Design and Implementation of the Sun Network File System,*" Proc. Usenix, Summer* 1985.

[22] "Trusted Tunis Integrity: Implementation of Integrity Controls Report #4", January 1990.

[23] Wall, L., R.Schwartz, *Programming Perl*, O'Reilly & Associates (pub), 1991.

[24] Wee, Christopher, "Protocol-directed Auditing," *Ph.D. Thesis (in preparation).*