Silicon Graphics

subject: **Extended Attributes: Plan G**

date: **February 10, 1995**

from: **Casey Schaufler**

ABSTRACT

The IRIX file system uses an attribute management scheme which is not extensible, each object having a small, fixed set of information kept about it. In order to implement mandatory access control (MAC), a mechanism is required to allow potentially large, variable sized information to be associated with these objects. This scheme implements extended attributes with a set of regular files in which the information is stored and a set of kernel functions which manipulate them.

Trusted IRIX/B - Extended Attributes: Plan G

1. Task Scope

2. Requirements

2.1 Orange Book 3.1.1.3.2.1

3.1.1.3.2.1 Exportation to Multilevel Devices

When the TCB exports an object to a multilevel I/O device, the sensitivity label associated with that object shall also be exported and shall reside on the same physical medium as the exported information and shall be in the same form (i.e., machine-readable or human-readable form). When the TCB exports or imports an object over a multilevel communications channel, the protocol used on that channel shall provide for the unambiguous pairing between the sensitivity labels and the associated information that is sent or received.

2.2 Other Requirements

Additional requirements for this task are detailed in **req.3.efs.mm**, **req.3.labels.mm**, and **req.3.trade.mm**. The security analysis for this task is detailed in **ana.3.efs_labels.mm**.

3. Security Functionality

The additional file attribute information required to implement MAC was the initial impetuous for an extended attribute mechanism. The possible future addition of access control lists inspired the search for a general scheme. Note that the "security functionality" provided by this mechanism is simply that there is a way to attach the new attribute information to the file system object.

4. Trusted Behavior

The example of MAC is used to describe how the plan G scheme is used.

4.1 Initialization

The *mount*(2) system call has been modified to call *eag_init*() twice, for `attribute/mac_index` and `attribute/mac_label`. The resulting inode pointers are stored in the mount table entry for that file system. If either call fails these pointers are set to NULL.

4.2 Getting Attribute Information

The kernel function *mac_access*() calls *eag_fetch*() twice, once for the inode to index mapping from the `mac_index` file, and once to get the label from the `mac_label` file. It then compares the label against the process label to determine the suitability of the requested operation. The same calls are made by the system call *getlabel*(2), which explicitly gets the label of a file for its caller.

4.3 Setting Attribute Information

The kernel function *com_nami*() calls *eag_set*() once or twice in each place that a file system object is created. It is always called to update the `mac_index` file, and may be called to update `mac_label` if the label is not already in that file. The same calls are made by the system call *setlabel*(2), which explicitly sets the label of a file for its caller.

5. Data Specification

5.1 Attribute Data Files

5.1.1 attribute/mac_label There is one of these files for each file system. This file contains MAC labels which have at some time existed on the file system. The format of each entry in the file is:

<Length><Label>

where *length* is a four-byte label length and *label* is the label itself. The length his kept here to allow swift searching of the file.

5.1.2 attribute/mac_index There is one of these files for each file system. This file contains the index of the label for a file in the `attribute/mac_label` file. The format of each entry in the file is:

<Index><Length>

where *index* is the byte offset in `attribute/mac_label` in which the label can be found and *Length* is the length of that label.

A performance optimization which should be considered is to store the label directly in the index file entry if it fits. If the high order bit of the index is set the eight bytes are assumed to be the label.

There is one entry for each disk-inode on the file system. The entry for the Nth inode is the Nth entry in the file.

5.2 Kernel Data

5.2.1 In-core Inodes The in-core inode is the data structure which is manipulated by the file system code. While the disk inode is of fixed size, the in-core inode varies depending on the number of extents required to represent the data contained by the file. What the disk inode keeps as indirect extents the in-core inode can keep directly.

5.2.2 The Label Lists Data for the four system special labels and three label lists are maintained by the Trusted IRIX/B kernel. There are separate lists for `MAC_TCSEC_LABEL`, `MAC_EQUAL_LABEL` and `MAC_MLD_LABEL` type labels.

5.2.3 The Mount Structure The `mount` structure for each file system is modified to contain pointers to in-core inodes for the `mac_index` and `mac_label` files. Accesses to these files can thus be made as needed by the kernel functions *eag_fetch*() and *eag_set*(). The in-core inodes are obtained by calls to *eag_init*().

5.2.4 The New In-core Inode A pointer to the label of the file is added to the in-core inode. This pointer points to an entry in the system label list.

5.2.5 NFS Support for the network file system protocol NFS falls out from the design. The only concern that must be addressed is that the daemons which run on behalf to the user on the server be granted access where needed. In particular, the daemons must run with appropriate labels and the code will have to uniformly use correct access control checks based on the daemon's capabilities.

6. Interface Specification

6.1 New System Calls

A user process may be allowed to read or write the label of a file using the *getlabel*(2) and *setlabel*(2) systems calls. The manual pages should be consulted for a more detailed description of the use of these systems calls.

These system calls use the kernel internal functions *eag_fetch*() and *eag_set*() rather than manipulating the files directly.

6.2 Within The Kernel

All manipulations of the plan G database files are done within the kernel. Direct manipulation of the database files by user processes is not supported and may have non-deterministic results.

6.2.1 eag_init The kernel internal function *eag_init*() is passed a path name in the kernel address space and the inode pointer of the directory to be used as the working directory. The specified file is opened and the resulting inode pointer returned. The resulting open file is not associated with the calling process, hence it is not closed should the process terminate. The function *copen1*() is called to do the bulk of the file open work.

6.2.2 eag_fetch The kernel internal function *eag_fetch*() is passed the number of bytes desired, their offset in the file, the inode pointer of the file, and the address in which to place the result. The file identified by the inode pointer is read into the passed address.

The actual I/O is done by a call to *FS_READI*().

6.2.3 eag_set The kernel internal function *eag_set*() is passed the number of bytes to be written, their offset in the file, the inode pointer of the file, and the address from which the data should be obtained. The file identified by the inode pointer is modified to contain the new information. The actual I/O is done by a call to *FS_WRITEI*().