SYSTEM V/MLS LABELING AND MANDATORY POLICY ALTERNATIVES

Charles W. Flink II and Jonathan D. Weiss

Charles W. Flink II *is a supervisor in the Information Systems Engineering Department of AT&T Bell Laboratories in Greensboro, North Carolina. Mr. Flink joined AT&T in 1983 and is responsible for the development of System V/MLS, a multilevel security version of the UNIX® operating system. He has a B.S. in physics from Loyola University and an M.S. in computer science from Virginia Polytechnic and State University.* ***Jonathan D. Weiss*** *is a member of technical staff in the Information Systems Engineering Department of Bell Laboratories in Whippany, New Jersey. Mr. Weiss joined the company in 1982 and is working on secure systems engineering for System V/MLS. He has a B.A. in mathematics and computer science from New York University and an M.S.E. in computer science from the University of Pennsylvania.*

System V/MLS is a product based on the UNIX® operating system, developed within AT&T Federal Systems Division to address National Computer Security Center (NCSC) requirements at the B level. This paper describes the alternatives that were analyzed in designing and implementing labeling and mandatory access control features to provide security, flexibility, and ease of use within a UNIX System V-compatible framework.

Introduction

System V/MLS was developed within AT&T Federal Systems Division (FSD) to meet customer demands for a computer operating system that can be evaluated by the NCSC. (System V refers to Version V of the UNIX system; MLS stands for multilevel security.)

The United States Department of Defense *Orange Book* specifies different levels of security for stand-alone computers; B-level requires extensive security features, and differs from the higher A-level in degrees of assurance.[1] For further discussion of B-level security, see the article by Barker and Nelson on security standards elsewhere in this issue of the *AT&T Technical Journal.*[2]

System V/MLS is designed to be evaluated at the B level, and to minimally affect the UNIX System V interface. Release 1.0 of the product meets this requirement as defined by *System V Interface Definition, Vol. 2,* (SVID) as tested by the *System V Verification Suite, Release 3.*[3]

In System V/MLS, we want to introduce security, while preserving (to the maximum extent possible) those features of the UNIX operating system that have made it successful. Certifiability at the B level demands a class of associations and restrictions that have not existed previously in the UNIX system. Incorporating *labeling* and *mandatory access control* (MAC) in a way that preserves a simple interface and maintains compatibility with UNIX System V is a major technical challenge. Labeling and mandatory access controls are applied to:

- *Subjects*—that is, processes
- *Objects*—including files, directories, i-nodes, interprocess communication (IPC) structures, and processes.

An i-node contains information about a file, including that file's permis-

Table I. UNIX Commands and Terms Used in This Paper*

Command	Description
`cat`	Concatenate and print files
`chmod` (1)	Change mode
`chown` (2)	Change owner or group
`cpio` (1)	Copy file archives in and out
`creat` (2)	Create a new file or rewrite an existing one
`/dev/null`	The null device
`/dev/tty`	File containing the user's terminal type
`environ` (5)	The user environment
`execute`	Set a process into motion
`id` (1)	Print user, group, fair share group IDs and names
`kill` (2)	Send a signal to a process
`link` (2)	Join users or directories together
`ln` (1)	Link files
`ls` (1)	List contents of directory
`mkdir` (2)	Make a directory
`mkgrp`	Create a new group
`mknod` (2)	Make a directory, file or special file
`mkpriv`	Create a new privilege
`namei` ()	Locate an i-node
`read` (2)	Read data from a file
`rm` (1)	Remove files
`rmdir` (2)	Remove directories
`search`	Enter or reference a directory
`sh` (1)	Shell, the standard command programming language
`spool`	A queueing facility
`stat` (2)	Get file status (i-node information)
`tar` (1)	Tape file archiver
`/tmp`	Temporary directory
`unlink` (2)	Remove links
`/usr/mail`	Mail directory
`/usr/spool/uucp`	Public directory containing files delivered via `uucp`
`utime` (2)	Change file modification and access times
`uucp`	UNIX system-to-UNIX system copy
`write`	Write data to a file

* Note: The numbers in parentheses refer to sections of the *UNIX System V, Release 3 User Reference Manual.*[4] For example, (1) refers to Section 1 of the *Manual.*

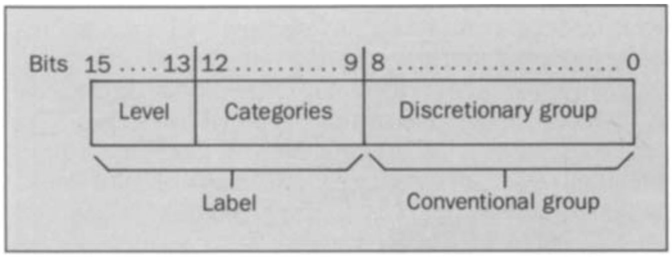

Figure 1. Subdivision of group identifier (GID) for direct labeling in the alpha release.

sions, disk address list, size, and type.

In this paper, we discuss design alternatives and decisions with respect to labeling and mandatory policy for System V/MLS. In evaluating design alternatives, we must consider:

- Certifiability
- Compatibility
- Simplicity of use and interface
- Flexibility.

Design issues related to other elements of System V/MLS are not addressed in this paper.

Labeling

We considered various options for the storage, representation, and association of subject and object labels. Based on customer requirements, we wanted a labeling approach that would be flexible and could be expanded to a number of specialized environments. In particular, we concluded that the NCSC minimum guideline that labels be capable of representing 16 hierarchical levels and 64 categories is insufficient for certain environments. In addition, some environments require labeling mechanisms that vary greatly from the notion of hierarchical levels and categories. Thus, flexibility is a primary consideration in our labeling scheme, particularly with respect to label storage, representation, association, and handling.

Whenever possible, we have parameterized label characteristics (that is, made labels dependent on system mappings and/or variables), provided ways to hide label representations from applications, and packaged mandatory policy in a separate module so that it would be easy to move the system to new application environments.

Label Storage. In the interest of compatibility and to expedite the porting of an identified set of applications, we were determined to create labeling in the UNIX system without modifying any underlying data structures. This allowed us to maintain not only an upwardly compatible system-call interface consistent with SVID but also a high degree of compatibility and interoperability with: protocol implementations; device drivers; established support procedures and organizations; conventional UNIX systems run in the system high mode; and conventional user training, practice, and experience.

Given current market realities, it is *not* reasonable or necessary, in our opinion, to establish an entirely new flavor of UNIX system with incompatible kernel-level interfaces, or incompatible backup/restore and file system structures, system programming, support personnel, etc. However, a departure from the existing structure of the UNIX system will be appropriate in the long-term evolution of this UNIX system product to address requirements at the higher levels of security defined by the NCSC. Certainly, major structural changes are required at the B3 level and beyond. Such restructuring will probably make most of the growing pool of device drivers and protocol modules obsolete and should not be taken lightly.

Because we wanted to add labels without modifying system data structures, we needed to identify an existing field in each applicable system data structure that could be used to contain label information without introducing gross incompatibilities. The *group identifier field* (GID) was chosen, based on the similarity of its characteristics to those of labels. In fact, it could be argued that the group identifier is the "natural" label for UNIX objects and subjects.

For example, there is a GID field for every subject as part of the process table entry and for every object as part of the i-node, IPC data structures, etc. Every object is "stamped" with the effective group identifier of the subject process that creates it. Child processes inherit the group identifier of their parent process. The system uses a policy that provides basic protection for this group "label" (although this policy had to be made mandatory rather than discretionary). Further, this group label is maintained by archiving utilities such as `tar` (1) and `cpio` (1), and displayed by utilities such as `ls` (1) and `id` (1).[4] (See Table I for definitions of UNIX system commands.) In UNIX systems that run in system high mode, groups are frequently used to separate various projects, and often directly or indirectly represent project sensitivity. Because these label-like characteristics of groups already exist, we decided to extend them to clearances and classifications.

Direct Labeling. Once we decided to use the group identifier for label storage, the next question was: How can we best use the 16 bits (or 65,536 unique GIDs) to represent sensitivity labels, *as well as* UNIX discretionary access control groups? A preliminary approach in the System V/MLS alpha release was to divide the group ID as shown in Figure 1.

This division allowed us to represent up to eight different hierarchical levels, four different categories and combinations thereof, and 512 different discretionary groups. The SVID dictates that 100 of these must be reserved for administrative groups, leaving 412 for users.

This labeling approach is simple; it has a minimal impact on the standard System V user interface, system calls, data structures, and kernel code; and it makes natural use of UNIX grouping.

This approach is limited, however, by the number of levels, categories, and discretionary groups as well as by its expandability (e.g., support of new fields) and flexibility.

The System V/MLS alpha release used this labeling scheme to demonstrate System V/MLS characteristics and interfaces to selected customer sites and to provide a base for experimental porting of applications. We included a library of label interface routines to hide the label representation so that we could move to the more flexible, more general solution: indirect labeling.

Indirect Labeling. The need for greater flexibility, as well as more levels, categories, and discretionary groups, prompted us to adopt an indirect labeling approach for our product. In this approach, the GID field (group identifier) no longer directly contains a label and discretionary group, but instead is an index into a data structure that contains this information.

Expandability. The GID-referenced data structure is not necessarily restricted to just sensitivity label and discretionary group; it may also contain a variety of other, less common protection mechanisms. These include discretionary caveats (handling instructions), special "privileges" granted to subjects marked with the GID, label ranges for multilevel devices or directories, and group mandatory/discretionary access control lists. (See Figure 2.)

Indirect GID-based labels offer:

- Minimal impact on standard UNIX System V user interface, system calls, data structures, and kernel code
- Natural use of UNIX grouping
- Many levels, categories, and discretionary groups
- Expandability
- Flexibility for specialized policies.

Disadvantages of indirect GID-based labels include:

- The limited number (65,536) of GIDs available
- Performance concerns and complexity of steps involved in accessing a label from the kernel level.

Neither disadvantage has proven to be significant in preliminary evaluation of the system. To address performance concerns, it is important to have a memory caching scheme for protection/authorization data structures; however, this is a minor effort compared to alternatives such as new file system structures.

Range limitations. Because the new data structure has no limits on the size of various fields, any number of levels, categories, and/or discretionary groups can be supported. However, the system may not have any more than 65,536 unique protection/authorizations (groupings). It is not likely that any one system would approach this limit. Objects are naturally grouped by users to be manageable.

For example, most files for a given project would share the same group designation. As a result, far fewer groups are needed than objects themselves. Few UNIX systems contain as many as 65,000 files, let alone *groups* of files. We expect that typical systems will use less than 1000 unique labels. The same tools that currently handle the movement of files between systems with different group designations can handle label translation for objects moved between systems with differing label definitions. As a result, the total universe of labels for any fixed set of meanings assigned to category bits and hierarchical levels

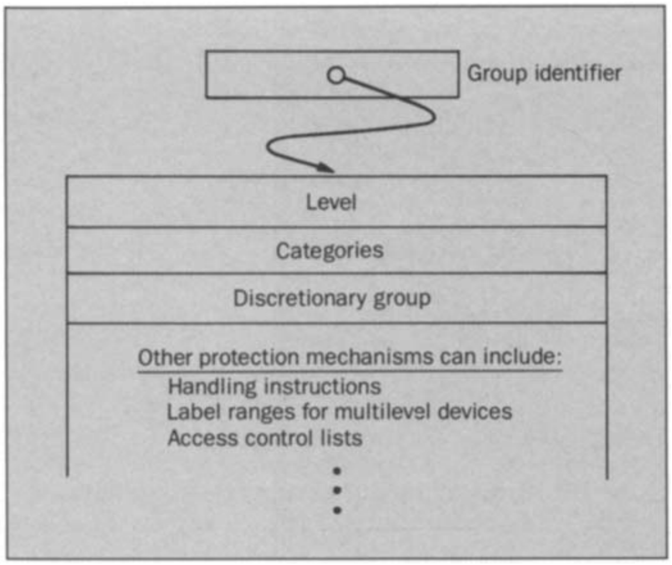

Figure 2. Indirect label via group identifier field (GID).

is *very* large. The limit of 65,536 unique labels only applies to a single machine.

Performance Impact. Access checks in the descriptor-based labeling environment are often only a check for identity between the group ID fields (descriptors) associated with the subject and object. This is because, in many cases, subjects access objects with the same inherited protections. For example, a subject (process) typically creates, and then accesses a set of temporary files with protections based directly on the subject's (process') authorizations. For such identity comparisons, there is no need for standard mandatory and discretionary access control checks.

Assurance Issues. The use of indirect labels and the problem of managing the association between GID and label seem to raise some assurance concerns. These have been carefully addressed in the design of the System V/MLS trusted computing base (TCB). First, the label to GID mapping in System V/MLS is immutable. Once a label is associated with a GID (when the GID is allocated via `mkgrp` or `mkpriv` commands), neither the label nor the associated GID can be changed by any activity short of intervention by trusted maintenance personnel. Further, a GID cannot be reused (assigned a new label) until after a certain interval of time has passed since any object marked with the GID has existed in the system. An administrator specifies the time interval (e.g., one year). If a GID is to be retired, all objects marked with the GID must be removed or reassigned before this "clock" starts. Secondly, the default label is "system high." If an object is imported or otherwise introduced to the system with a GID not mapped in the label file, the system grants no access to the object until the object is assigned a known GID (and hence label) by a trusted administrator.

An initial concern raised during the evaluation of indirect labels for the System V/MLS product was that the association between a GID and a particular label is host-environment-dependent and, thus, insufficient as an exportable label. Note, however, that labels with explicit numbers for levels and bits for categories (i.e., the direct approach described above) are also environment-specific with respect to exportation. The particular numerical levels and individual bits only have meaning in the specific environment in which their mappings are defined.

For instance, the bit corresponding to a particular category can mean "NATO" (North Atlantic Treaty Organization) on one machine and "NOFORN" (no foreign nationals) on another. Even if the bit represents the "same" concept across machines (e.g., NOFORN), the actual meaning of the bit can be different depending on the nation in which the system is operated. As a result, whenever *any* label representation is exported, it must be a procedural requirement that its meanings (or mappings) be exported as well. Thus, the indirect label approach suffers no added risks when compared with direct labeling.

Label Handling. With indirect labeling and a label interface library, label representation for System V/MLS is hidden and isolated. We further isolated the interpretation of (and operations performed on) labels in a kernel-level module known as the *MLS module*.

The MLS module is a removable and replaceable portion of our system that allows us to adapt and evolve our labeling scheme and security policy to track customer needs. It is done by inserting "hooks" in various UNIX system kernel routines that call the appropriate functions

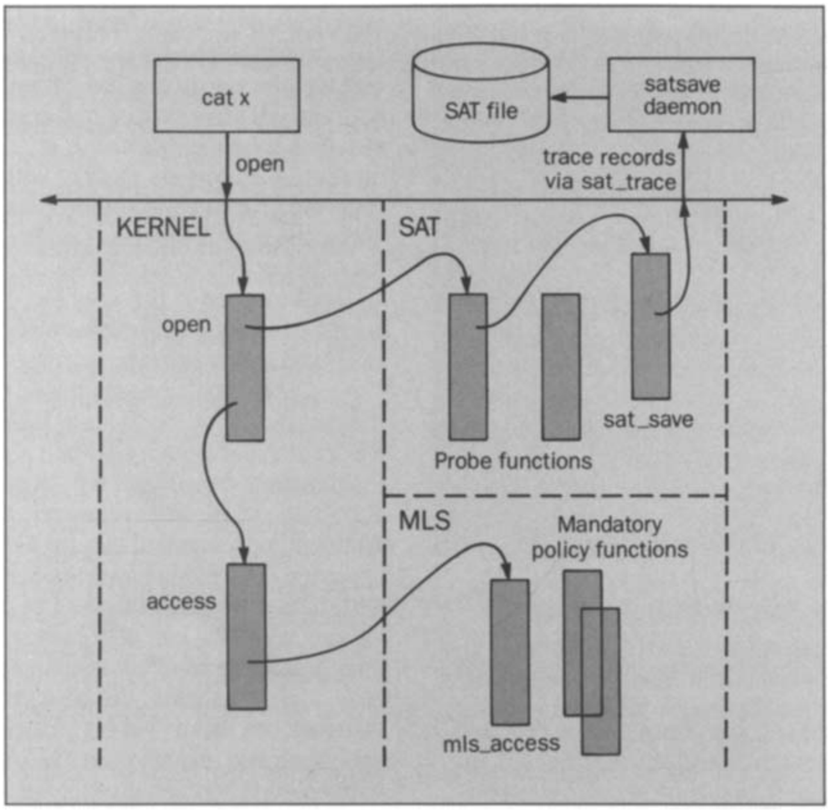

Figure 3. System V/ MLS modules.

in the MLS module when policy decisions must be made. A similar structure has been established for the System V/ MLS *security audit trail* (SAT module).

Figure 3 shows the interaction between a UNIX kernel routine and functions within the MLS (and SAT) modules. The figure shows the calling sequence for enforcement of access controls (and access auditing) for the `read` (2) that results from the `cat` (concatenate) command.

The label representations, and the rules applied, are not visible in the kernel routines. They are solely the concern of the MLS module.

Project interface. System V/MLS supports an enhanced version of the standard UNIX system discretionary access controls. Although this article focuses on labeling and *mandatory* policy, it is important to discuss briefly the connection between System V/MLS labels and its *discretionary access control* (DAC) mechanism.

System V/MLS has a "project-oriented" DAC interface that allows selected individuals to set up and manage projects. A project is simply a discretionary group in the conventional UNIX sense. Because it may be defined at multiple clearance and classification levels, there may be more than one GID and group file entry for any given project. For example, a project "shuttle" can be defined in System V/MLS and associated with permissible clearances/classifications (e.g., Top-Secret, or Secret-NATO, or Confidential-NATO-Crypto). With such associations established, subjects and objects may operate only within the project if they are cleared at one of the allowable levels. Likewise, the clearance levels of subjects and objects may be changed only to an allowable level within

Table II. Mandatory Policy Notation

Symbol	Represents
R	Read (file, directory, etc.)
S	Search (directory)
E	Execute
W	Write (overwrite or append)
O	Overwrite
A	Append
C	Create [`creat` (2), `mkdir` (2), `mknod` (2), etc.]
L	Link [`ln` (1), `link` (2)]
U	Unlink [`rm` (1), `unlink` (2), or `rmdir` (2)]
St	Read file/i-node status [`stat` (2), etc.]
Ch	Change status [`chown` (2), `chmod` (2), `utime` (2), etc.]
K	Send a signal [`kill` (2)]
Ripc	Read IPC mechanism
Wipc	Write (alter) IPC mechanism
O	The object in question
Od	Directory object or directory containing object
Of	Simple file object (not directory)
S	Subject (process acting on user's behalf)
>=	Label of left item "dominates" label of right
==	Identical sensitivity label

their current project.

Placement of users in various clearance levels of a project is controlled by the project administrator. (The project administrator can be simply the "owner" or first member of the group.) A project administrator can grant or deny membership in a project (consistent with user clearances) at the user level. By setting the proper UNIX system protection bits [via `chmod` (1)], an *object owner* may further decide to grant read, write, and/or execute access to him/herself, all users within the object's current project, or everybody. Thus, one can think of projects as shared access-control lists that are administered by project managers, but can be switched on and off by object owners. If an existing project does not provide the proper discretionary protections for a file, a new protection may be easily created by the project administrator or another authorized individual. Intervention by the system administrator (or large-scale granting of *super user* privileges) is not required.

Mandatory Policy Alternatives

Before any discretionary checks, the System V/MLS mandatory access control (MAC) policy is applied. In creating policy alternatives, we looked at mandatory protection of:

- Files
- I-nodes
- Directories
- IPC mechanisms (and their associated data structures)
- Processes (as recipients of signals).

With each alternative, we tried to map the simple security (ss-) and *-properties of the Bell-LaPadula security model into a UNIX system framework.[5] (The ss-property specifies "no read up;" the *-property specifies "no write down.")

Table II shows the UNIX system operations involved in enforcing mandatory controls; it also defines the notation used to compare alternative policies.

For IPC mechanisms, the `Ripc` and `Wipc` symbols in Table II represent a class of operations. All IPC operations can be mapped into one of these two classes. (See the *UNIX System V, Release 3 Programmers' Manual.*[6]) The `Od` and `Of` symbols are used to distinguish between checks on a file-type object versus its parent directory. The meanings of the other symbols will be explained in more detail as they are encountered.

File Protections. At first glance, neither the "no read up" (ss-property) nor the "no write down" (*-property) rules seemed to place restrictions on writing *up* for files. Panel 1 shows the file protection alternatives we considered.

Policy 1 was not an acceptable solution because it violated a "no destroy up" principle that seemed to us to be fundamental, although it is not a requirement in the *Orange Book.*[1] We therefore modified our file protection policy as shown in Policy 2 of Panel 1.

This solved the "no destroy up" problem, but left a covert channel (a *-property violation) in the form of the error codes returned when an attempt is made to `append` to a file of strictly dominating classification. For example, a high-level user can turn the discretionary `write` permissions on a high-level file on and off, thereby signaling a low-level user making multiple attempts to `append` the file. This problem can be solved by doing "blind" `appends` (success code always returned). However, we deemed this an unreasonable and undesirable mechanism for real-world systems. We chose Policy 3 (shown in color in Panel 1).

Because this policy no longer allows `write` "up" in any form, the covert channel discussed above is not an issue. This alternative is more restrictive than others, and forces upward communication through other means (i.e., upward reclassification). For integrity reasons, we decided it was the superior approach, because it prevents low-level users from affecting objects at higher levels.

Panel 1. File Protection Alternatives
Several alternatives were considered; Policy 3 (shown in color) was selected.

Policy 1

Operations	Dominance Relation
R/E W(O/A)	S >= O O >= S

Policy 2

Operations	Dominance Relation
R/E O A	S >= O S == O O >= S

Policy 3

Operations	Dominance Relation
R/E W(O/A)	S >= O S == O

I-node Protections. Having obtained a reasonable policy for protecting file information, we examined i-nodes. (See Panel 2.) We noticed that a parallel existed between operations that queried information in the i-node and file `reads`, and operations that altered information in the i-node and file `writes`. This led us to the rules shown in Policy 1 of Panel 2.

Here, `stat` (St) and `change` (Ch) are the primary i-node observation and alteration functions, and `link` (L) and `unlink` (U) are included because they change the link-count field of the i-node.

This policy seemed reasonable and allowed no obvious covert channels, but was inconsistent with the current structure of standard UNIX system protections. In

Panel 2. I-node Protection Alternatives
Two alternatives were considered; Policy 2 (shown in color) was selected.

Policy 1

Operations	Dominance Relation
St	S >= O
Ch/L/U	S == O
U (last link)	S >= O

Policy 2

Operations	Dominance Relation
St	S >= O
Ch	S == O

the current UNIX system, directory permissions are used to determine `link` and `unlink` capabilities for objects within them. There is the matter of access times in lower-level classified objects being updated by `reads` by higher level subjects. This is a classic example of a covert channel. The ultimate solution will require a redefinition of the access time feature. The alternatives are either not to update the access time on `reads` from higher level subjects or not to report it via the `stat` (2) system call. The expedient alternative is to document this as a covert channel and throttle as necessary (e.g., `stats` after `reads` from above will be delayed *X* seconds).

We concluded that it would be perfectly valid for the label on the parent directory to be the basis for the controls on `link` and `unlink` operations. Applying the parent directory label for `link` and `unlink`, and the object label itself for `stat` controls, can create a covert channel only when the object is classified at a lower level than the parent. This may happen in System V/MLS only when an object is declassified by a security administrator (or locally authorized individual). Therefore, we chose to make prevention of the covert channel a procedural issue for declassification, rather than an i-node access protection issue. Policy 2 (shown in color in Panel 2) was the result.

Directory Protections. Having decided to use parent directory labels for `create`, `link`, and `unlink` controls, we considered alternative policies for directory protections. (See Panel 3.)

Policy 1 was attractive for its flexibility and partial resolution of a class of problems related to temporary directories. The UNIX system relies on many directories [e.g., public directories such as `/tmp` (or temporary) and `spool`] that, with the introduction of labeling, become storage places for objects with multiple classifications. In conventional UNIX systems, temporary directories represent a problem because the discretionary rule that allows any subject to create a file in the directory also allows any other subject to remove the file. With the introduction of labels, there is the extended problem of establishing rules to allow a directory to have files of numerous levels that would still be appropriately protected. With the directory protection policy shown in Policy 1 of Panel 3, such directories would be labeled system high and files of all levels could be stored in them. The rule for `unlink` would assure that no higher level information could be removed by a lower level user, and the rule for `search` would permit users to access the objects for which they were authorized in the higher level directory.

However, the `search` rule allows lower level users to test for the existence of higher level objects. Even though the rule for directory `read` prevented the lower level subject from listing a directory, if the subject knew the name of the object, it could attempt to access the object and note success or failure. This situation represented a covert channel. The problem can be partially corrected by issuing the same return code for failure to satisfy the mandatory controls as for "object not found;" however, it would still be possible to check for the existence of classified directories by attempting to `search` them. We could reduce this problem, but decided that customers are less interested in the flexibility provided in this approach than in the additional security provided by the approach shown in Policy 2.

This policy is certainly stricter than the earlier one. Objects may only be `created`, `linked`, and `unlinked` from directories whose labels are identical to that of the subject. Furthermore, a directory's contents may only be detected by subjects whose labels dominate. The only way that an object's label may differ from that of the parent directory is if the object is created and then reclassified in an authorized way. (For System V/MLS, all subjects can upwardly reclassify objects that they own, while designated security administrators can declassify.)

With this policy, however, we do have the problem of temporary directories. Because of the identity rule for `create/link`, there is no way under the stated policy to classify directories to permit creation of objects of multiple levels. This means that, without some special corrective measure, all current applications using temporary directories would no longer run.

For the alpha release of System V/MLS, we modified the policy as shown in Panel 3 until a solution to the temporary directory problem could be found.

This policy was unacceptable because it allowed users to create higher-level classified file/directory names in lower-level directories. [The problem of shared (e.g., `/tmp`) directories is discussed below.]

IPC and Signals. The policy for IPC mechanisms and signals is shown in Panel 4. There is a problem here because IPC objects share a common, unprotected name space. A Trojan horse (a form of malicious code executed inadvertently) operating at a classified level can signal an accomplice by creating or deleting IPC objects in the common name space. Several alternative solutions for this problem are under consideration.

The primary question for these policies was whether signals could be sent to processes whose labels dominate the subject's label. We decided that it was better to protect the higher level processes from being affected by the lower level ones.

Temporary Directory Alternatives. Many solutions for the temporary directory problem have been proposed including:
- Label ranges for directories
- Subjective directories
- Cloning devices
- Parameterized symbolic links.

Panel 3. Directory Protection Alternatives
Several alternatives were considered; Policy 2 was selected, but was modified for the alpha release.

Policy 1

Operations	Dominance Relation
R	S >= O
S	(no mandatory check)
C/L	Od >= S
U	Od >= S and S >= Of

Policy 2 (Selected)

Operations	Dominance Relation
R/S	S >= O
C/L/U	S == Od

Modified Policy - Alpha Release

Operations	Dominance Relation
R	S >= O
C/L/U	S >= Od

Label ranges. One way to solve the problem is to apply similar controls to directories and multilevel devices (i.e., to have a maximum and a minimum label associated with the directory). This lets a range of classified objects be placed in the directory, with removal controls based on the object label itself. Because this solution is not consistent with the selected mandatory policy, or the current philosophy of UNIX system access checks, it was rejected in System V/MLS.

Subjective directories. Another solution is to alter the `namei()` routine to recognize special directories flagged as "subjective." For these special directories, `namei()` can be modified to insert the subject's current sensitivity label (suitably encoded) into the target path, invisibly changing all temporary directory references to

Panel 4. Signal and IPC Protections

Selected Policy

Operations	Dominance Relation
Ripc	S >= O
Wipc	S == O
K	S == O

Panel 5. System V/MLS Protection

Mandatory Policy Summary

Operations	Dominance Relation
R/S/E	S >= O
W(O/A)	S == O
C/L/U	S == Od
St	S >= O
Ch	S == O
Ripc	S >= O
Wipc	S == O
K	S == O

refer to a suitably labeled (and hence protected) subdirectory.

This proposed solution is called *subjective directories* because the directory given to the user is not the one strictly identified by the path. Instead, it is a different directory selected by the sensitivity level of the subject. The "objective" or true view of the file tree is available only to administrative processes. This mechanism may be applied not only to the temporary directory problem, but also to directories like `/usr/mail` and `/usr/spool/uucp` to implement multilevel mail and `uucp` with limited changes to these subsystems.

By applying this meaning to the currently unused directory "set-GID" bit, directories can be recognized by `namei()` as being subjective. It may also be desirable to establish subjective directories parameterized by UID (user identifier) as well, using the "set-UID" bit as a flag.

System V/MLS uses the "subjective directory" approach to the temporary directory problem. However, instead of appropriating the directory "set-ID" bits, which might be applied to some other purpose in future UNIX System V releases, it flags the directory with the special group "SECURED." When the `namei()` function encounters a directory with this special group, it performs the substitution described above.

Cloning devices. An alternate approach to solving the temporary directory problem can be based on a pseudo-device or new file-system type that clones a private, appropriately labeled, secure directory for every appropriate new access to the actual directory. The new file system or cloning device can be mounted over `/tmp`, `/usr/tmp`, and any other directory for which subjective-directory-like functions are needed. Such a solution requires additional effort to implement, and offers no advantages over the subjective directory approach.

Parameterized symbolic links. A final alternative was an extension of the concept of the symbolic link. The symbolic link of UNIX System Research Version 8 or 4.2 BSD (Berkeley Software Distribution) is simply an absolute path that is substituted for the path being interpreted by `namei()`. If the symbolic link is extended to include variables in `sh` (1) (shell) notation, which will then be expanded by `namei()` from the environment of the process [`environ` (5)], a highly flexible solution results. The files in `/tmp` and `/usr/tmp` could be symbolic links to `/tmp.d/ $LEVEL` and `/usr/tmp.d/$ LEVEL`, where `LEVEL` is an environment variable, that is set when entering a classified level. The `LEVEL` variable is then set to the classification level just entered. As a result, the symbolic link can be made to point to an appropriately classified subdirectory, segregating temporary files based on classification and thereby protecting them.

Further, this technique can be generalized to any other example of a shared resource directory. For example, `mail` (1) can be easily used as a multilevel secure mail system if the directory `/usr/mail` is a parameterized symbolic link to `/usr/securemail/$ LEVEL`. Invoking `mail` from a classified level will automatically access a suitably labeled, classified mailbox that corresponds to the level in question. Similar examples can be provided for such file uses as `lp` (1), `/dev/tty`, and `/dev/null`, simplifying the implementation of mandatory protection for these resources as well. If symbolic links are included in future releases of UNIX System V, this may become the best alternative for solving the temporary directory problem.

Policy Summary

The full, selected policy is shown in Panel 5. We believe it provides the proper combination of flexibility and security for the various access operations. Because the mandatory policy is localized in a limited set of MLS module routines, it can evolve to future requirements. Thus, it can be made more flexible *or* more secure depending on customer needs.

Many additional elements not discussed here influence the security policy of the system (e.g., level changing commands, declassification). They are described in the informal System V/MLS security model.

Conclusion

We believe that the chosen labeling scheme and mandatory protection policy can be taken beyond the Department of Defense B1 security certification level to meet requirements at higher certification levels. In creating labeling and mandatory policy without embedding assumptions about label format or meaning within the kernel itself, System V/MLS provides a degree of flexibility allowing implementations that can go well beyond the standard *Orange Book* mandatory protection model.[1] We intend to maintain the principles of flexibility, simplicity, and compatibility throughout the product's evolution.

Note

The product described in this paper is not to be interpreted as a new *standard* release of AT&T System V, nor should it be considered a statement of how the *System V Interface Definition* will necessarily evolve to address NCSC security requirements.[3] The product is available from the AT&T Federal Systems Division as an *enhanced security* version of UNIX System V, Release 3. It is being sold to government customers and vendors under the understanding that it cannot be guaranteed to be fully "upwardly compatible" with (as yet unspecified) future AT&T, national, or international standards for secure systems. It will be sold and supported by FSD until standards are established and AT&T System V evolves to meet the special security requirements of the government systems marketplace.

References

1. "DoD Trusted Computer Systems Evaluation Criteria," United States Department of Defense, No. 5200.28, December 1985.
2. L. K. Barker and L. D. Nelson, "Security Standards—Government and Commercial," *AT&T Technical Journal*, Vol. 67, No. 3, May/June 1988, pp. 9-18.
3. *System V Interface Definition*, Vols. 1 & 2, AT&T Customer Information Center, Indianapolis, Indiana, 1986.
4. *UNIX System V, Release 3 User Reference Manual*, AT&T Customer Information Center, Indianapolis, Indiana, 1987.
5. D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, MTR-2997, Revision 1, MITRE Corporation, Bedford, Massachusetts, March 1976.
6. *UNIX System V, Release 3 Programmer's Manual*, AT&T Customer Information Center, Indianapolis, Indiana, 1987.

(Manuscript received April 14, 1988)