A FORMAL MODEL FOR UNIX SETUID

Tim Levin, Steven J. Padilla, and Cynthia E. Irvine

Gemini Computers, Inc.
P.O. Box 222417, Carmel, CA 93922

A B S T R A C T

The UNIX setuid mechanism is described in the context of the GEMSOS architecture. Motivation for modeling setuid is given. Modeling and policy requirements for the control of the setuid mechanism are presented. The GEMSOS formal security policy model is compared with the Bell and LaPadula model(1). The Bell and LaPadula model is shown to not admit the actions of a setuid mechanism. Features of the GEMSOS DAC model are described that represent the actions of the UNIX setuid mechanism while limiting their negative effect on the DAC policy.

INTRODUCTION

Gemini is engaged in the development of a UNIX-compatible (2) interface to be supported by an underlying TCB of class A1 as described in the Trusted Computer System Evaluation Criteria [TCSEC]. Gemini has chosen an architecture for its secure products which uses hierarchical protection domains based on protection rings [SCHRO] to separate security-relevant from non-security-relevant functions. These domains are used to further subdivide security-relevant functions into those enforcing mandatory access controls (MAC) and those enforcing discretionary access controls (DAC). Non-security-relevant functions are separated into distinct protection domains so that an operating system such as the UNIX-compatible interface is protected from less-privileged application software.

The GEMSOS TCB is implemented using Intel IAPX286 hardware, which provides four hardware-enforced privilege levels. The GEMSOS mandatory security kernel always occupies hardware privilege level zero. GEMSOS maps eight hierarchical rings into the three remaining hardware privilege levels. Thus, at any instant a process can include up to three active subjects (where a subject is considered to be a process, ring pair), each associated with a different privilege level.

Table 1 describes the GEMSOS architecture. The boundaries between the six rows represent the separation of major subsystems. Mandatory access controls are enforced by the GEMSOS security kernel. MAC support, which includes, for example, multilevel subjects and login, is confined to rings 1 and 2. DAC policy is enforced by the GEMSOS Discretionary TCB, the outer boundary of which extends to ring 4. The TCB does not extend beyond ring 4.

In our design, the UNIX-compatible interface occupies ring 5. This operating system will provide much of the functionality of a standard UNIX kernel; however, security-related features such as identification and authentication are within the TCB. Applications software occupies the two least privileged rings in the system, rings 6 and 7.

---

1. In this paper, "Bell and LaPadula model" refers to the Multics Interpretation of that model [BLP].

2. UNIX is a trademark of American Telephone and Telegraph Co.

TABLE 1. GEMINI ARCHITECTURE

System Component	Protection Ring	Comments
Application	6 - 7	Commercial-Off-The-Shelf or other application-specific software
Gemini UNIX-Compatible Interface	5	an operating system providing a UNIX interface
GEMSOS Discretionary TCB	3 - 4	provides discretionary access controls implementing the GEMSOS DAC model
GEMSOS Non_Kernel MAC Support	1 - 2	support functions for the GEMSOS mandatory security kernel
Sysgen	0	system generation (must be able to write kernel-ring segments)
GEMSOS Security Kernel	0	provides mandatory access controls implementing the GEMSOS MAC model

The GEMSOS architecture is in contrast to various monolithic approaches to a UNIX-compatible interface in which all UNIX functionality is encompassed by the TCB. A limitation of these monolithic architectures is that there are many features supported by standard UNIX interfaces which do not enforce security, including: the complete set of UNIX file attributes, the complete set of UNIX process attributes, starting and stopping spoolers, and chronometric information. TCSEC standards for B3 and A1 systems require the minimization of the TCB to include only security-relevant features, not a complete range of operating system functionality. Therefore, in these systems of the highest assurance, the TCB interface cannot be the full UNIX interface.

A distinguishing characteristic of the UNIX operating system is the "setuid" mechanism [SYS5][RITCH]. UNIX associates a user id, the process "real id," with every user process as a result of the user successfully completing a login. Permissions to access objects in UNIX are calculated relative to another id, the process's "effective id." Upon successful completion of login, the real id and effective id are both the authenticated user id. The setuid mechanism allows an active process to change its effective id by executing certain program objects. The result of this change is for the process to gain discretionary permissions different than those it had before the execution of the object. This also breaks the tie between the user and the subject acting on the user's behalf, since the subject is no longer associated (for DAC permission decisions) with the user's authenticated id.

In order for the GEMSOS TCB to support a UNIX implementation, special attention must be given to UNIX's "setuid" and "setgid" capabilities. The setuid capability, which modifies the user identification for a process, is described below. Setgid is similar, but sets the group identification of a process, and will not be described in detail. Two complementary mechanisms comprise setuid for standard UNIX systems.

The CHMOD and EXEC function calls allow the setuid mechanism to be invoked when certain code files are executed. Each executable code file has an owner, a setuid mode bit, and a setgid mode bit. CHMOD may be used to set the user and group "set_id" bits on a program object. When the EXEC call is used to execute the object, if either the user or group set_id bit is set, then the executing process assumes (as its effective id) the user or group id of the owner of the object.

An explicit SETUID function call is also provided for invocation from an executing application process. The SETUID call is used to change the effective id of the process back to its real id. When the setuid function call is invoked, the effective user id of a process is changed to the value passed by the call if:

A. the effective user id of the process is superuser (In this case the real user id of the process is also changed to the value passed by the call.), or

B. the real user id of the process matches the user id passed by the call, or,

C. in many UNIX implementations, a saved user id matches the user id passed by the call.

The UNIX setuid mechanism is intended to be used to allow users access to system-sensitive information while restricting

the scope of their operations on that information. Because the setuid mechanism leaves the real ids of processes unaffected, and the system audit functions are based on real ids, setuid has no impact on audit functions. The remainder of this paper will describe Gemini's approach to modeling the implicit setuid mechanism.

MODELING AND POLICY REQUIREMENTS FOR CONTROL OF THE UNIX SETUID MECHANISM

Constraints on the GEMSOS model and policy were imposed by the requirements of the TCSEC, as interpreted by NCSC evaluation team for the GEMSOS TCB, and are discussed later in this section. In order to constrain changes of id, the team indicated that changes in id should only be allowed when associated with an object. Currently, this constraint precludes the use of the SETUID call, while allowing both CHMOD and EXEC. This report does not address the modeling of the SETUID call. Inclusion of the setuid function would require both additional modeling work and modification of the constraints described herein. Use of the SETUID call is still being studied and may be included following additional clarification from the team.

The setuid mechanism provides a means for associating permissions (e.g., read or write) with predefined (programmed) activities, somewhat similar to a software implementation of protected subsystems [SALTZ][BUNCH]. This association can be used to provide an application-specific set of permissions to objects in a given domain. These permissions can be viewed by those invoking the subsystem as access modes to object types, both of which are defined by the subsystem creator.

However, since the Bell and LaPadula model has both current access set and permission matrix indexed by subject, there is a problem modeling a state change such as that induced by the setuid mechanism. Abstractly, setuid changes a process's permissions. This can be modeled either as a change in permission matrix entries for the subject representing the process, or a change in the subject representing the process (thus resulting in a whole different row of permissions in the permission matrix). If setuid is considered to change the entries of the permission matrix, there is a problem mapping this change to a system where permissions are represented by Access Control Lists or access bits (e.g., UNIX), since the execution of the setuid program results in no corresponding change to the set of ACLs or access bits.

On the other hand, if setuid is considered to be a change in the subject representing the process, then there is a problem of consistency between the current access set and the permission matrix. Since there is a different subject representing the process in the permission matrix, that new subject must represent the process in the current access set. However, there is no change in the current accesses of the process as a result a setuid operation. This implies that the new subject representing the process has the same current accesses as the old subject, i.e., that there is a set of subjects representing the process with identical current accesses. However, there is no model mechanism linking such subjects together, and the get-access rule of the Multics Interpretation only changes current access for one subject.

In order to model the benefits of the setuid mechanism (i.e., UNIX compatibility and software enforced protected subsystems) while minimizing its negative effect on DAC policy (i.e., breaking the tie between the authenticated user id and the effective user id), we represent and restrict the setuid-mechanism semantics in the GEMSOS DAC model. These restrictions take the form of three modeling and four "policy" requirements:

Modeling:

1. The process shall be represented by a different subject after the process changes DAC id.

2. DAC ids shall be associated with objects for support of the setuid mechanism.

3. "Control" mode is added to the list of access modes.

Policy:

1. A process shall only be able to assume the DAC id that has been associated with an object.

2. A process shall only assume such an id if it has current "execute" access to said object (the discretionary access property assures that the process must have had permission to get current access).

3. The DAC id associated with an object shall only be changed to the calling process's id, or some portion thereof if ids have more than one aspect (e.g., user and group).

4. The DAC id associated with an object shall only be changed if the calling process has "control" access to the object.

It has also been suggested that there be some control shown over the entry point for the object executed under the setuid mechanism. While this is protection relevant, it is at a level of detail beyond that described by the access-control policy and models discussed here. The initial "setuid" entry point in the GEMSOS TCB is strictly controlled and this control is specified in the formal top level specification (FTLS), but not in the GEMSOS DAC model. The UNIX implementation of entry points for setuid objects can be supported with the GEMSOS TCB.

MODELING OVERVIEW

The TCSEC requires that a formal security policy model be produced for TCB systems in classes B2 through A1. This model must include a mathematically precise statement of the abstract security policy enforced by the TCB and it is used to both articulate the policy and to prove that the TCB supports the policy.

The Bell and LaPadula model is a widely used formal security policy model. The GEMSOS model (composed of a MAC model [LEVIN] and a DAC model) is shown to derive from the basic principles of the Bell and LaPadula model. The GEMSOS DAC model discussed herein is in the process of being mapped to the GEMSOS TCB and is not final.

The UNIX operating system enforces a DAC policy, but does not control information flow (i.e., does not enforce a MAC policy). Therefore, only the DAC portion of a formal security policy model is necessary to describe the security policy of UNIX.

As described in the previous section, the UNIX "setuid" mechanism allows users to use program actions to gain permissions to objects. This feature has an undesirable effect on the enforcement of DAC policy: it breaks the authenticated link between users and the subjects that act on their behalf.

In the following sections, the essential features of the GEMSOS and Bell and LaPadula models are compared, the GEMSOS DAC model is described, and specific elements of the GEMSOS DAC model used to represent and control setuid are discussed.

COMPARISON OF THE BELL AND LAPADULA AND GEMSOS MODELS

The Bell and LaPadula model is a mathematically formulated state transition model that represents the control of active subjects' access to passive objects. The model consists of elements, properties, and rules. The elements represent the state of the model. The properties represent the global restrictions on state (for example, the system security policy is defined in the properties). The rules define the state changes for a particular valid interpretation (e.g., an implementation) of the model.

Elements

Elements of the Bell and LaPadula and GEMSOS models include subjects, objects, modes of access to the objects, and labels. Further, elements are included to represent the relation of labels to each other, the relation of labels to subjects and objects, the hierarchy of objects, the access to objects by subjects, and the permission to objects for subjects.

The Bell and LaPadula model has "trusted subjects" which are implicitly trusted from subject-max-level ("read_class") to system low. In contrast, the GEMSOS model adds a minimum ("write_class") label, which is always dominated by the read_class label, for all subjects. In GEMSOS, subjects are trusted when the read_class and write_class labels are not equal; they are untrusted when the read_class and write_class labels are equal. As a result of the inclusion of the write_class label instead of system low, a GEMSOS trusted subject may have more limited access to objects than a Bell and LaPadula trusted subject. (Note that the concept of limiting the range of "write" trust for a trusted subject has subsequently been adapted in other presentations [BELL][LEE].)

The elements of the GEMSOS model are represented in the Ina Jo [SCHEI] specification language (3) as follows (some of these elements are modified to represent setuid, below):

```
TYPE
   subject,
   object,
   access = (execute,read,append,write),
   label
CONSTANT
   obj_label(object) : label,
   dominates(label, label): boolean
VARIABLE
   curr_access(subject,object,access)
                : boolean,
   permission(subject,object,access)
                : boolean,
   parent_child(object,object)
                : boolean,
   read_class(subject) : label,
   write_class(subject) : label
```

Properties

Properties define the desired security characteristics for the system. The two properties defined for the Bell and LaPadula model are the mandatory security property and the discretionary security property. As stated in terms of elements from the GEMSOS model, the mandatory security property states that in order to perform a read operation, the subject's read_class label must dominate the object's label (called "simple security"), and in order to perform a write operation, the subject's write_class label must be dominated by the object's label (called "confinement").

The discretionary security property, as enunciated in the Bell and LaPadula model states that if a subject has current access to an object, the subject must also have permission to the object. This means that when permission to an object is revoked, current access to the object must also be revoked, immediately. However, once a subject has access to an object, the subject can copy the object, thus retaining permission to the information through its permissions to the copy. Recognizing this, the GEMSOS model modifies the discretionary security property to require that when NEW current access is gained, the subject has permission to the object. The effect of this change is that the revocation of a permission is not necessarily reflected immediately in a corresponding change to current access. Since the Bell and LaPadula model, in effect, allows permission to the information after permission to the object has been revoked, the GEMSOS model does not reflect a weaker policy with respect to information flow.

When the model properties (which define security) are enforced relative to the rules (which define state changes), the "basic security theorem" is proven to hold inductively in the context of the state model: given a secure initial state, the preservation of security from one state to the next guarantees system security.

These two basic properties are represented in the GEMSOS model as follows, using the notation of Appendix A (some elements of these properties are modified to represent setuid, below):

```
CRITERION /* Mandatory security property */
    a" s:subject,o:object(
       (  curr_access(s,o,read)
        | curr_access(s,o,write)
       -> dominates(read_class(s),
                    obj_label(o))
       )
       &
       (  curr_access(s,o,append)
        | curr_access(s,o,write)
       -> dominates(obj_label(o),
                    write_class(s))
       )
    )
CONSTRAINT /* Discretionary Security
                  Property */
    a" s:subject,o:object,a:access(
       ~curr_access(s,o,a)
       &
       n"curr_access(s,o,a)
       ->
              permission(s,o,a))
```

The initial conditions of the GEMSOS model are such that the criterion and constraint are true. Consider "true" an Ina Jo shorthand for this expression.
INITIAL true

Rules

The specific rules of the Bell and LaPadula model and the GEMSOS model are not discussed in this report since they are both relative to specific systems (Multics and GEMSOS, respectively). However, it is important to understand that regardless of the specific rules defined for a model, the proof that the model rules enforce the security properties demonstrates that the basic security theorem holds for that model.

---

3. See Appendix A for an overview of the Ina Jo Language syntax.

A SEPARATE DAC MODEL

A Mandatory Access Control policy defines classes of information and describes the allowed (or conversely, the disallowed) flows between the classes. Typically, the classes are partially ordered. A well known example of a MAC policy is the classification system used in many sectors of the government. Under this policy, information is classified Top Secret, Secret, Confidential, or is Unclassified (thus establishing classes for information) and the permitted flow of information is defined:

```
Unclassified -> Confidential
        -> Secret -> Top Secret
```

where "->" is defined to mean "may flow to."

A Discretionary Access Control policy creates domains (in the sense of a mathematical function) of identified objects based on accesses permitted (to those objects) for identified users. Each user has a domain in which authorization (e.g., read or write) is granted to all objects in that domain. Users access system objects by way of active system entities (subjects). The authorization characteristic of DAC policy requires that there be a strong tie between individual users and the subjects that act on their behalf. This tie, referred to as an "unforgeable id" [SALTZ], is typically created by some form of identification and authentication (I&A) mechanism. Since an identification and authentication policy is not an access control policy it is not usually addressed in a formal security policy model. Interpretation of the model does however assume the existence and proper enforcement of such a supporting policy. In the GEMSOS DAC model, a "DAC id" is included to help describe how the tie between users and subjects is created and maintained.

Recognizing that the DAC and the MAC policies can be described independently and are separately enforceable [SHOCK], the GEMSOS model has been split into separate MAC and DAC models. This is advantageous when working with the UNIX system which does not deal with labels, but is only concerned with discretionary security. Thus, the model for UNIX need not be encumbered with the dominates and label-relationship elements, nor with the mandatory security property. As Shockley points out, in the GEMSOS system, MAC is enforced in a layer beneath the DAC layer. This presents an overall enforcement mechanism that ensures that all system references are constrained by the properties of both DAC layer and the underlying MAC layer. The MAC properties are described in the MAC model and the DAC properties are described in the separate DAC model.

The elements and properties of the separate GEMSOS DAC model are represented as follows (some of these elements are modified to represent setuid, below):

```
TYPE
    subject,
    object,
    access=(execute, read, append, write),
    label
VARIABLE
    curr_access(subject,object,access)
                : boolean,
    permission(subject,object,access)
                : boolean,
    parent_child(object,object)
                : boolean
CONSTRAINT
    a" s:subject,o:object,a:access(
       ~curr_access(s,o,a)
       &
       n"curr_access(s,o,a)
       ->
           permission(s,o,a))
INITIAL true
```

This model is sufficient to support a restricted UNIX discretionary access mechanism, i.e., one without the setuid mechanism. For example, the UNIX protection bits of (RW.RW....) for an object represent that the owner of the object and all members of his group have both read and write permission to the object, but that the public has no access to the object. These permissions can be represented in this model with a true values in all "permission" entries containing the subjects representing the owner and group members, the specified object, and the modes of read and write.

DAC MODEL ELEMENTS TO CONTROL THE UNIX SETUID MECHANISM

To represent the setuid mechanism in the model and enforce the stated modeling and policy requirements, the following elements are included in the GEMSOS DAC model.

```
TYPE
    id,
    subj_eq_class,
    dac_id = structure of (
        user  = id,
        group = id,
        env   = id
    ),
    subject = structure of(
        equiv_class= subj_eq_class,
        dac_access = dac_id
    ),
    access = (execute,read,append,
              write,control)

VARIABLE
   active_subj_id(subj_eq_class):dac_id,
   caller:subj_eq_class,
   dac_attribute(object):dac_id,
   curr_dac_attribute(subj_eq_class,object)
                      :dac_id,
   permission(dac_id,object,access)
                      :boolean,
   curr_access(subj_eq_class,object,access)
             :boolean
```

The model elements ID and DAC_ID are added to the model in order to associate a "user-oriented" id with the active subjects of the model. The USER, GROUP and ENV fields of DAC_ID are provided to add granularity to the notion of ID. These allow the model to incorporate the policy notions of user and group, and allow a separate "environment" aspect to be distinguished.

The notion of a system "process" is modeled as an equivalence class of subjects (SUBJ_EQUIV_CLASS); when changing its DAC_ID, the process changes subjects within its equivalence class. The current access set (CURR_ACCESS) is indexed by SUBJ_EQUIV_CLASS. This shows how the model representation of a process consists of a class of subjects, each with the same current access set. Thus, in a given state the SUBJ_EQUIV_CLASS is the set of all subjects that may become active within the process.

The variable ACTIVE_SUBJ_ID indicates the current DAC_ID and (implicitly the "active" subject) for each equivalence class of subjects. The model's PERMISSION matrix is indexed by DAC_ID. The change of index shows how a SUBJ_EQUIV_CLASS's permissions change after changing DAC_ID.

The variable CALLER is added as a global element. This allows the model to identify the calling process, independent of the transform statement, and is useful in stating global constraints.

The DAC_ATTRIBUTE of an object associates a DAC_ID with an object.

When the object is added to the current access set (CURR_ACCESS) of a SUBJ_EQUIV_CLASS, the DAC_ATTRIBUTE is copied to the per-equivalence-class/per-object variable element, CURR_DAC_ATTRIBUTE. When a SUBJ_EQUIV_CLASS (representing a process) executes the object under the setuid mechanism, it assumes the DAC_ID stored in its CURR_DAC_ATTRIBUTE, for that object. This means that a change to the global DAC_ATTRIBUTE does not change the equivalence class's CURR_DAC_ATTRIBUTE, i.e., a the per-equivalence-class version is not revoked with the global attribute.

And finally, "CONTROL" mode is added to the list of model accesses. This mode is used to control which subjects can change the DAC characteristics of objects (as opposed to the information contained in the objects).

In the context of the UNIX setuid mechanism, the "CONTROL" mode is necessary to allow control of the CHMOD function presented at the UNIX interface. Recall that using CHMOD, the owner of a file can modify its access mode so that the setuid bits are set, with the consequent change of effective user id to that of the file's owner when the file is executed.

DAC MODEL PROPERTIES TO CONTROL THE UNIX SETUID MECHANISM

The following properties are included in GEMSOS DAC model to implement the policy requirements stated above. These properties do not preclude building a UNIX-compatible setuid mechanism based on this model. The Ina Jo representation of the GEMSOS DAC model (including the properties discussed below) is listed in Appendix B.

The Current DAC Attribute Property

The Current DAC Attribute Property ensures that when a SUBJ_EQUIV_CLASS (process) gets process-local access (CURR_DAC_ATTRIBUTE) to an attribute of an object, the attribute (DAC_ID) is the DAC_ATTRIBUTE of the object. This property controls which ids a process can switch to using the setuid mechanism.

```
CONSTRAINT /* Current DAC Attribute
              Property */
   a"eq:subj_eq_class,obj:object(
       n"curr_dac_attribute(eq,obj)
         ~= curr_dac_attribute(eq,obj)
       &
       n"curr_dac_attribute(eq,obj)
         ~= null_dac
       ->
         n"curr_dac_attribute(eq,obj)
           = dac_attribute(obj)
   )
```

Assume ID Property

Assume ID Property ensures that in order to assume a different DAC_ID, the process must have current "execute" access (CURR_ACCESS) to an object whose "CURRENT_DAC_ATTRIBUTE" contains (the changed portions of) that DAC_ID. In order to get this current access, the SUBJ_EQUIV_CLASS must have had permission to execute the object. This property controls which processes can switch ids using the setuid mechanism.

```
CONSTRAINT /* Assume ID Property */
   a"eq:subj_eq_class(
       n"active_subj_id(eq)
         ~= active_subj_id(eq)
       ->
          e" obj:object(
              curr_access(eq,obj,execute)
              &n"active_subj_id(eq).user =
                 (curr_dac_attribute(eq,obj)
                                    .user
                           = null_id
                 => active_subj_id(eq).user
                 <>
                  curr_dac_attribute(eq,obj)
                                    .user
                 )
              & n"active_subj_id(eq).group =
                 (curr_dac_attribute(eq,obj)
                                    .group
                           = null_id
                 =>
                  active_subj_id(eq).group
                 <>
                  curr_dac_attribute(eq,obj)
                                    .group
                 )
              & n"active_subj_id(eq).env =
                 (curr_dac_attribute(eq,obj)
                                    .env
                           = null_id
                 => active_subj_id(eq).env
                 <>
                  curr_dac_attribute(eq,obj)
                                    .env
                 )))
```

Dac Attribute Property

Dac Attribute Property ensures that if the DAC_ATTRIBUTE of an object is changed, it is only changed to the caller's ACTIVE_SUBJECT_ID (or a portion thereof). The NULL_ID is used to indicate that the DAC_ATTRIBUTE of the object will not effect the given portion (USER, GROUP, ENV) of the calling process's DAC_ID, when a new DAC_ID is assumed through the setuid mechanism. This property controls which ids can be associated with an object for the setuid mechanism.

```
CONSTRAINT /* Dac Attribute Property */
   a"obj:object(
       n"dac_attribute(obj)
       ~= dac_attribute(obj)
       ->
          (n"dac_attribute(obj).user
                = null_id
           |
           n"dac_attribute(obj).user
                = active_subj_id(caller)
                                .user
          )
          &
          (n"dac_attribute(obj).group
                = null_id
           |
           n"dac_attribute(obj).group
                = active_subj_id(caller)
                                .group
          )
          &
          (n"dac_attribute(obj).env
                = null_id
           |
           n"dac_attribute(obj).env
                = active_subj_id(caller)
                                .env
          )
   )
```

Discretionary Control Property

The Discretionary Control Property ensures that the DAC_ATTRIBUTE for an object can only be changed if the calling process has CONTROL access to the object. (The GEMSOS DAC model also ensures that the calling process must have current write access to the parent of the object.) This property controls which processes can set the ids associated with an object for the setuid mechanism.

```
CONSTRAINT /* Discretionary Control
              Property */
   a"o:object(
     n"dac_attribute(o) ~= dac_attribute(o)
     -> permission(active_subj_id(caller),
                   o,control)
        &
        e"parent:object(
           parent_child(parent,o)
           &
           curr_access(caller,parent,write)
        ))
```

CONCLUSIONS

The GEMSOS formal security policy model is divided into mandatory access control (MAC) and discretionary access control (DAC) portions. The DAC portion (GEMSOS DAC model) includes elements and properties to show that systems represented by the model exert control over the setuid mechanism, that is, the subjects of the model cannot arbitrarily change their permissions or (consequently) their current accesses.

The GEMSOS DAC model is formulated to be compatible with UNIX setuid and is shown to:

1. allow the model to reflect a change of subject when a process changes user-id,
2. restrict the processes that can use the setuid mechanism,
3. restrict the id's that are assumable through the setuid mechanism,
4. associate an id with objects to support the setuid mechanism
5. restrict which processes can set the id associated with a given object, and
6. restrict the id's that can be set on the object.

REFERENCES

[BLP] Bell, D.E. and LaPadula, L.J. , "Computer Security Model: Unified Exposition and Multics Interpretation," Tech. report ESC-TR-75-306, MTR-2997 Rev.1, The Mitre Corporation, Bedford, Mass., March 1976

[BELL] Bell, D. Elliot, "Security Policy Modeling for the Next-Generation Packet Switch," in Proceedings of the 1988 IEEE Symposium on Security and Privacy, pp. 212-216, Oakland CA, April 18-21, 1988

[BUNCH] Bunch, Steve, "The Setuid Feature in UNIX and Security," in Proceedings of the 10th National Computer Security Conference, September 1987, pp. 245-253

[LEE] Lee, Theodore M.P., "Using Mandatory Integrity to Enforce "Commercial" Security," in Proceedings of the 1988 IEEE Symposium on Security and Privacy, pp. 140-146, Oakland CA, April 18-21, 1988

[LEVIN] Levin, T. "Formal Security Policy Model for the GEMSOS Kernel," GEMINI, Technical Report in preparation.

[RITCH] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System," The BELL System Technical Journal, Vol. 57, No. 6, July-August 1978

[SALTZ] Saltzer, J.H., and Schroeder, M.D., "The Protection of Information in Computer Systems." In Proceedings of the IEEE, Vol. 63, No. 9, September 1975, pp. 1278-1308.

[SCHEI] Scheid,J., Anderson, S., Martin, R., and Holtzberg, S., "The Ina Jo Specification Language Reference Manual--Release 1." TM 6021/001/02. System Development Corporation, Santa Monica, Ca., 1986.

[SCHRO] M. Schroeder and J. Saltzer, "A Hardware Architecture for Implementing Protection Rings," Commun. A.C.M., vol 15, pp. 157-170, Mar. 1972.

[SHOCK] W.R. Shockley and R.R. Schell, "TCB Subsets for Incremental Evaluation", in Proc. 3rd Aerospace Computer Security Conference, 1987, American Institute of Aeronautics and Astronautics, Washington, D.C.

[SYS5] System V Interface Definition, Select Code No. 320-013, AT&T Customer Information Center, Indianapolis, Ind, 1986.

[TCSEC] Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD, December 1985.

Appendix A. Overview of the Ina Jo Specification Language Syntax

Notation	Comment/Meaning
a"x:TYPE	Universal quantifier: for all x of type TYPE
e"x:TYPE	Existential quantifier: there exists an x of type TYPE
n"	New-value operator
nc"	No-change operator
constraint	global property between successive states
criterion	global property for all states (top level only)
invariant	global property for all states
/* ... */	comment
<>	Else
=>	Then
->	Implies
|	Or
&	And
~	Not
=	Equals

Appendix B. Ina Jo Representation of the GEMSOS DAC Model

The elements and properties of the GEMSOS DAC model are presented below.

```
SPECIFICATION gemsos_dac
LEVEL model

TYPE
    id,
    label,
    object,
    subj_eq_class,
    dac_id = structure of (
        user  = id,
        group = id,
        env   = id),
    subject = structure of(
        equiv_class= subj_eq_class,
        dac_access = dac_id),
    access = (execute,read,append,
              write,control)

CONSTANT
    null_id:id,
    null_dac:dac_id

VARIABLE
    active_subj_id(subj_eq_class):dac_id,
    caller:subj_eq_class,
    curr_access(subj_eq_class,object,access)
            : boolean,
    curr_dac_attribute(subj_eq_class,object)
            : dac_id,
    dac_attribute(object):dac_id,
    parent_child(object,object)
            : boolean,
    permission(dac_id,object,access)
            : boolean
INITIAL true

CONSTRAINT /* Discretionary Security
              Property */
    a" eq:subj_eq_class,o:object,a:access(
        ~curr_access(eq,o,a)
        & n"curr_access(eq,o,a)
        ->
          permission(active_subj_id(eq),o,a))

CONSTRAINT /* Current DAC Attribute
              Property */
    a"eq:subj_eq_class,obj:object(
            n"curr_dac_attribute(eq,obj)
              ~= curr_dac_attribute(eq,obj)
            & n"curr_dac_attribute(eq,obj)
              ~= null_dac
            ->
               n"curr_dac_attribute(eq,obj)
               = dac_attribute(obj))

CONSTRAINT /* Assume ID Property */
    a"eq:subj_eq_class(
        n"active_subj_id(eq)
        ~= active_subj_id(eq)
        ->
          e" obj:object(
             curr_access(eq,obj,execute)
             &
             n"active_subj_id(eq).user=
                (curr_dac_attribute(eq, obj)
                                    .user
                          = null_id
                 => active_subj_id(eq).user
                 <>
                  curr_dac_attribute(eq,obj)
                                     .user)
             &
             n"active_subj_id(eq).group =
                (curr_dac_attribute(eq, obj)
                                   .group
                          = null_id
                => active_subj_id(eq).group
                <>
                  curr_dac_attribute(eq,
                                     obj)
                                    .group)
             & n"active_subj_id(eq).env =
                (curr_dac_attribute(eq, obj)
                                    .env
                          = null_id
                 => active_subj_id(eq).env
                 <>
                  curr_dac_attribute(eq,
                                     obj)
                                    .env)))
```

```
CONSTRAINT /* Dac Attribute Property */
   a"obj:object(
       n"dac_attribute(obj)
       ~= dac_attribute(obj)
       ->
          (n"dac_attribute(obj).user
                = null_id
           |
           n"dac_attribute(obj).user
                = active_subj_id(caller)
                                .user)
          &(n"dac_attribute(obj).group
                = null_id
           |
           n"dac_attribute(obj).group
                = active_subj_id(caller)
                                .group)
          &(n"dac_attribute(obj).env
                = null_id
           |
           n"dac_attribute(obj).env
                = active_subj_id(caller)
                                .env))

CONSTRAINT /* Discretionary Control
              Property */
   a"o:object(
     n"dac_attribute(o) ~= dac_attribute(o)
     -> permission(active_subj_id(caller),
                   o,control)
        &e"parent:object(
          parent_child(parent,o)
          &
          curr_access(caller,parent,write)))

END model
END gemsos_dac
```