A Discussion Of Formal Models

Frank Knowles
Gould, Inc.
Computer Systems Division - Urbana
1101 East University Avenue
Urbana, ILL 61801
U. S. A.

GOULD
Electronics

Nov 15, 1985

A DISCUSSION OF FORMAL MODELS

This talk is divided into two parts.

The first part is an exposition of the Bell - La Padula (B-L) model. The model referred to is that described in the MITRE report, "Unified Exposition and Multics Interpretation." This model has been translated into a VERUS specification, and VERUS versions of some model constructs are presented.

The second part is a sketch of a model (not formal) for a B1 level UNIX.

The B-L model is the most discussed security model for operating systems, so it is appropriate to start with it if one is interested in writing a model for a secure system. In a sentence, the B-L model is a state machine model for an operating system with four access modes (read, write, read-and-write, execute), eight generic state transformations (give-permission, rescind- permission, get-access, release-access, create-object, delete-object, change-current-subject-level, change-object-level), and three state invariants (simple-security, star-security, discretionary-security).

The model is not intended to address covert channels or system integrity, nevertheless it is a reasonable first step in describing those aspects of an operating system that could result in the undesirable transfer of information.

Some aspects of the model are unclear. Giving and taking of permissions is incompletely specified and the rules regarding permissions are stated differently in different parts of the paper.

The rules for accessing objects in a hierarchy are unsatisfactory. They require a user to change security levels after creating an object in order to gain read-and-write access to it.

The proposed model for a B1 level UNIX limits transfer of information between objects of different classification levels in the spirit of the B-L model, but equal emphasis is given to trusted subjects and access restrictions that guarantee system integrity.

BELL - LA PADULA SECURITY MODEL

CONTENTS

THE BELL - LA PADULA MODEL

This is an exposition of the Bell - La Padula (B-L) model. I will describe the underlying state machine model, the elements of the model, the security policy, and the transformation rules. The exposition is based upon the MITRE report, "Unified Exposition and Multics Interpretation."

There are some differences between the exposition in the first three papers on the model and the fourth. For instance, there are change-level rules, and discretionary access is incorporated into the security requirements. Nevertheless, I will pretend that there is just one model and it is the "unified" one.

I will also identify points of incompleteness or ambiguity and problems that will arise if the model is applied to a real system.

1 INTRODUCTION

The B-L model is the best known abstract security model for operating systems. It is a reasonable first step in analyzing a system for unwanted data flow. As the authors say in their informal discussion, the model does not address covert channels or system integrity. The primary goal seems to be to present a set of rules that disallow a direct transfer of data from a storage object at a high level to a storage object at a lower level.

As a preparation for this talk, the model was recast in the VERUS language and each rule was proven to be security-preserving. Some examples of VERUS declarations and proofs will be presented.

2 SUBJECTS AND OBJECTS

The model is concerned with access of subjects (users, processes, etc.) to objects (files, memory segments, etc.). The set of subjects may overlap the set of objects.

Subjects are of two types, trusted and untrusted. In general, trusted subjects are subject to lesser checks than untrusted subjects when requesting an access.

3 TYPES OF ACCESS

There are four types of access of subject to object

execute — no observation or alteration of the object

Standard interpretation is "run" for a data file and "search" for a directory.

read — observation of the object only

Standard interpretation is "read" for a data file and "list" for a directory.

append — alteration of the object only

Standard interpretation is "write" for a data file and creation or deletion of files for a directory. The name is an unfortunate choice.

write — both observation and observation of the object

Standard interpretation is "read and write" for both data files and directories. The name is another unfortunate choice.

4 OBJECT HIERARCHIES

Objects are assumed to be arranged in an hierarchy. A hierarchy is the usual directory structure limited to directed graphs. That is, links that yield a circular path are outlawed and children cannot have more than one parent.

These details of object structure are largely irrelevant as far as the proofs of security go except when it comes to deleting an object--all objects below are affected. The compatibility principle (see below) is a relationship between parent and child, but it is not mentioned in the security policy (see below), hence it is not security-relevant.

5 ACCESS MATRIX

The current activity in the system can be be visualized as a matrix with each row representing a subject and each column representing an object.

The entry for (subject, object) is the set of current accesses being requested (and presumably being acted upon) by the subject for the object.

6 PERMISSION MATRIX

As we shall see, one aspect of a "secure" state is that all accesses be "permissible" accesses. Those accesses that are permissible are represented by a permission matrix similar to the (current) access matrix.

Each row represents a subject, and each column represents an object, and each entry is a set of accesses, just as in the access matrix. The entry, (subject, object), in this case, is the set of "permissible" accesses of subject to object.

7 SECURITY LEVELS

Each subject and each object has associated with it a security level (or label). A subject has both a maximum level and a current level. The maximum level must always dominate the current level. The exact structure of the levels is not important in the security proofs as long as certain axioms about the levels hold.

The set of security levels forms a partially ordered set. That is, we can talk about L1 >= L2 (L1 dominates L2), or L1 is not comparable to L2, where "dominates" satisfies three axioms:

1. L1 >= L1
2. L1 >= L2 and L2 >= L1 implies L1 = L2
3. L1 >= L2 and L2 >= L3 implies L1 >= L3

A level will actually consist of a "classification" and a set of "compartments." For instance, a level might be the tuple, (secret, NATO, Nuclear). In this example, "secret" is the classification and the set consisting of the other elements, NATO and Nuclear, is the set of compartments. The first component of the level comes from a totally ordered set where all elements are related (none are non-comparable). Such a set might look like:

top secret, secret, confidential, unclassified

where "top secret" is greater than "secret"; "secret" is greater than "confidential", etc.

The second component of the level is a set of compartments. These elements do not have any relation to each other. They are intended to represent a "need to know" requirement. Such a set might look like:

NATO, Nuclear, CRYPTO

Now we can say how two security levels can be compared. L1 >= L2 means that the classification component of L1 is greater than or equal to that of L2 and the set of compartments of L1 includes the set of compartments of L2.

Two levels can be non-comparable if the level of one is greater than or equal to that of the second but the set of compartments of the second is not a subset of the set of compartments of the first. For example, these two levels are not comparable:

(top secret, NATO) and (secret, NATO, Nuclear).

It is easy to see that the set of levels as we have defined them satisfies the requirements of a partially ordered set as given above. From now on, we can forget how the levels actually look.

The declarations in VERUS of some of these system components look like this:

```
$ TYPE DECLARATIONS
$ Entity includes both subjects and objects.

TYPE Entity;
TYPE Subject OF Entity;
TYPE Object OF Entity;

TYPE SecurityLevel;

$ Declare access modes.

TYPE Access = (e_access, r_access, a_access, w_access);

$ Declare access elements.

TYPE Elt;                   $ Declare access elements as a type.
TYPE EltSet;                $ We will also need sets of access elements.

$ SET MACROS

$ Use the SetDefs macro to the pull in the definitions of
$ set operations for sets of elements.

SetDefs( Elt, EltSet )

$ This is part of the output of the SetDefs macro applied to elements:

$ Declare "is a member of" primitive for sets of elements.

DECLARE InEltSet(Elt, EltSet) : BOOLEAN;

$ Axiom of set union.

DEFINE  UnionEltSet(set1EltSet, set2EltSet) : EltSet
BY
    FOR elt1Elt
      EQUIV
      {
        InEltSet(elt1Elt, UnionEltSet(set1EltSet, set2EltSet));
        OR
        {
          InEltSet(elt1Elt, set1EltSet);
          InEltSet(elt1Elt, set2EltSet);
        };
      };
$ Besides the basic operations, axioms for set equality and the
$ empty set are generated.
```

```
$ MORE TYPES

$ Each Elt is a triple, (Subject, Object, Attribute):

DECLARE SubjectOf( Elt ) : Subject;
DECLARE ObjectOf( Elt ) : Object;
DECLARE AttributeOf( Elt ) : Access;

$ Declare the current access set.

DECLARE CurrentAccess( Time ) : EltSet;

$ Declare the access permission matrix as a function
$ of subjects and objects.

DECLARE Permissions( Time, Subject, Object ) : AccessSet;

$ Declare security level functions.

DECLARE MaxLevel( Subject ) : SecurityLevel; $ maximum level
DECLARE ObjectLevel( Time, Object ) : SecurityLevel;
DECLARE CurrentLevel( Time, Subject ) : SecurityLevel;  $ current level

$ Declare the "dominates" relation for security classes.

DECLARE Dominates( SecurityLevel, SecurityLevel ) : BOOLEAN;

$ Define some notions associated with the hierarchy of objects.

DECLARE Parent( Time, Object, Object ) : BOOLEAN;

VAR obj, obj1, obj2 : Object;

DEFINE Root( t, obj ) : BOOLEAN
BY
  FOR obj1 NOT Parent( t, obj1, obj );

$ This is a recursive definition of obj1 being inferior to obj2:

DEFINE Inferior( t, obj1, obj2 ) : BOOLEAN
BY
  OR
  {
    Parent( t, obj2, obj1 );
    EXIST obj
     AND
     {
       Inferior( t, obj, obj2 );
       Parent( t, obj, obj1 );
     };
  };
```

8 UNDERLYING STATE MACHINE MODEL

I will not follow the presentation of the underlying state model as it is presented in the report. It is awkward and more complicated that it needs to be. Instead, I will present the state machine model as it is usually formalized in VERUS.

The underlying state machine consists of the "state" (the collection, with current values, of subjects, objects, access matrix, permission matrix, and security level functions described earlier), a state requirement (a conjunction of statements that must be true of the initial state and all succeeding states), and a set of state transformations (rules). I will specify the state requirement in the next section. Let's call a state that satisfies the state requirement, a "secure" state.

The burden of presenting a state machine model consists of defining all the things just alluded to and of proving that the initial state satisfies the state requirement and that each transformation sends secure states into secure states. Thus, all reachable states are secure.

We need to define an initial state of the state. In the model the following state is suggested. The Initial State consists of a set of subjects, a set of objects, a permission matrix, the three level functions previously defined, and an EMPTY access matrix. Since there are no current accesses, it follows easily from the state requirement in the next section that this is a secure state.

To complete the definition of the B-L model we need to define the state requirement, and the transformation rules.

9 SECURITY POLICY

This section defines the state requirement or "security policy" of the model. A "secure" state is a state in which the following three things are true.

For the set of current accesses, both the simple security property and the star property must hold, and between the entries in the access matrix and the permission matrix, the discretionary control requirement must hold.

A precise statement expressed in VERUS looks like this:

```
$ STATE REQUIREMENT

$ Declare the security properties of the system.

DECLARE SimpleSecurity( Time ) : BOOLEAN;
DECLARE StarProperty( Time ) : BOOLEAN;
DECLARE DiscretionarySecurity(Time) : BOOLEAN;

DEFINE StateRequirement( t ) : BOOLEAN
BY
  AND
  {
    SimpleSecurity(t);
    StarProperty(t);
    DiscretionarySecurity(t);
  };

$
$ Define the simple security condition.
$ Note: InEltSet is defined in SetDefs.1.
$ It is the relationship of set membership between
$ access elements and sets of access elements.
$
DEFINE SimpleSecurity(t) : BOOLEAN
BY
  FOR elt
    IF AND
    {
      InEltSet( elt, CurrentAccess(t) );
      ObserveAccess( elt );
    }
    THEN Dominates( MaxLevel(SubjectOf(elt)),
                    ObjectLevel( t, ObjectOf(elt)));
```

```
$ Define the *-property.

$ If Trusted (subj) Then subj is a trusted subject.

DECLARE Trusted( Subject ) : BOOLEAN;

DEFINE StarProperty(t) : BOOLEAN
BY
  FOR elt
    IF AND
    {
      InEltSet( elt, CurrentAccess(t) );
      NOT Trusted( SubjectOf(elt));
    }
    THEN AND
    {
      IF AttributeOf(elt) = a_access
      THEN Dominates( ObjectLevel(t,ObjectOf(elt)),
                      CurrentLevel(t,SubjectOf(elt)));

      IF AttributeOf(elt) = w_access
      THEN ObjectLevel(t,ObjectOf(elt)) = CurrentLevel(t,SubjectOf(elt))

      IF AttributeOf(elt) = r_access
      THEN Dominates( CurrentLevel(t,SubjectOf(elt)),
                      ObjectLevel(t,ObjectOf(elt)));
    };

$ Define discretionary security.
$ Note: InAccessSet is similar to InEltSet except that it denotes
$ set membership between access attributes and sets of access
$ attributes.

DEFINE DiscretionarySecurity(t) : BOOLEAN
BY
  FOR elt
    IF InEltSet( elt, CurrentAccess(t) )
    THEN InAccessSet( AttributeOf( elt ),
                      Permissions( t, SubjectOf(elt), ObjectOf(elt)));
```

10 COMPATIBILITY

In B-L, "compatibility" means that the level of the parent of an object is dominated by the level of the object.

I suppose that this is called "compatibility" because otherwise you can create objects that, in normal circumstances, you wouldn't be able to access. The motivation for this requirement is discussed on page 29 of the report. The argument is that having

L < level (D)

would mean that U could never write F. The reason is that to access any file in D, U must have execute access to D so level of U >= level of D, but to write F, level of U <= L, which is < level of D!

11 TRANSFORMATION RULES

This section contains a description of the state transformation rules. In the model, these rules are described in three different places. Firstly, they are given a multics flavor using segment field values and diagrams. Secondly, they are stated with mathematical fanfare. Thirdly, they are stated informally. Finally, proofs are given that each rule preserves security.

I will state each rule but, except for one, I will skip the proof that the rule preserves security. All of the proofs are easy to derive informally, and only one or two cause any difficulty when using VERUS. I will illustrate the VERUS proof style by showing a simple proof.

11.1 Get_Read.

Get_Read (S, O) is a request that subject S obtain read access to object O.

This request is granted if the following is true:

The permission matrix shows that S may have read access to O

AND

the maximum level of S dominates the level of O AND either S is trusted or the current level of S must also domininate the level of O.

If the request is granted then an access of S to O in read mode is added to the current access matrix.

This seems to imply that trusted subjects are subject to discretionary access control.

11.2 Get_Append.

Get_Append (S, O) is a request that subject S have append access to object O.

This request is granted if the following is true:

The permission matrix shows that S may have append access to O

AND

either S is trusted or the level of O domininates the current level of S.

If the request is granted then an access of S to O in append mode is added to the current access matrix.

Notice that untrusted subjects may "write up."

11.3 Get_Execute.

Get_Execute (S, O) is a request that S have execute access to O.

This request is granted if the following is true:

The permission matrix shows that S may have execute access to O

If the request is granted then an access of S to O in execute mode is added to the current access matrix.

Seems odd to me that there is no mandatory check for execute.

11.4 Get_Write.

Get_Write (S, O) is a request that subject S have write access to object O.

This request is granted if the following is true:

The permission matrix shows that S may have write access to O

AND

the maximum level of S dominates the level of O AND either S is trusted or the current level of S equals the level of O.

If the request is granted then an access of S to O in write mode is added to the current access matrix.

11.5 Release_Access.

Release_Access(S, O, A) means that S releases access to O in mode A.

Here is the VERUS specification of this rule.

```
VAR x_access : Access;
CONST Cx_access : Access;
CONST Cx_access2 : Access;
CONST Celt : Elt;

DEFINE ReleaseAccess( subj, obj, x_access ) : BOOLEAN
BY
  AND
  {
    FOR elt
      IF AND
      {
        SubjectOf(elt) = subj;
        ObjectOf(elt) = obj;
        AttributeOf(elt) = x_access;
        InEltSet( elt, CurrentAccess(OLD) );
      }
      THEN NOT InEltSet( elt, CurrentAccess(NEW))
      ELSE
        IF NOT InEltSet( elt, CurrentAccess(OLD) )
        THEN NOT InEltSet( elt, CurrentAccess(NEW))
        ELSE InEltSet( elt, CurrentAccess(NEW));


    $ We also need to specify that no other state functions change:

    SamePermissions;
    SameCurrentLevel;
    SameObjectLevel;
    SameParent;
    SameHierarchy;
  };
```

Here is the VERUS proof that Release_Access preserves security.

```
$ Declare instantiation constants.

CONST Csubj, Csubj1, Csubj2, Csubj3 : Subject;
CONST Cobj, Cobj1 : Object;

PROVE
  IF AND
  {
    StateRequirement(OLD);
    EXIST subj EXIST obj EXIST x_access
         ReleaseAccess( subj, obj, x_access );

  }
  THEN StateRequirement( NEW );
{
  PROVE SimpleSecurity( NEW );
  {
    EXIST subj EXIST obj EXIST x_access
         ReleaseAccess( subj, obj, x_access );

   SimpleSecurity( OLD );
  };

  PROVE StarProperty( NEW );
  {
    EXIST subj EXIST obj EXIST x_access
         ReleaseAccess( subj, obj, x_access );
    StarProperty( OLD );
  };

  PROVE DiscretionarySecurity( NEW );
  {
    EXIST subj EXIST obj EXIST x_access
         ReleaseAccess( subj, obj, x_access );
    DiscretionarySecurity( OLD );
  };
};
```

11.6 Give_Permission.

Give_Permission (S1, S2, O, A) means that subject S1 gives permission to subject S2 to have access to object O in mode A.

Comment: There are different definitions of this rule in the report. I will give the slightly simpler one. GIVE(S1, O) means that S2 may give permissions to O. This is not further defined in the report.

This request is granted if the following is true:

O is not the ROOT, and S1 has current write access to the Parent of O

OR

O is the ROOT, and GIVE (S1, O)

If the request is granted then the permission matrix will now show that S2 may have access to O in mode A.

11.7 Rescind_Permission.

Rescind_Permission(S1, S2, O, A) means that S1 rescinds permission for S2 to access O in mode A.

Comment: RESCIND(S, O) means S may rescind permissions for access to O. This function is not defined further in the model.

This request is granted if the following is true:

O is not ROOT, and S has write access to Parent(O)

OR

O is ROOT, and RESCIND(S, ROOT);

If the request is granted then A is removed from the entry (S2, O) in the permission matrix, and any access by S2 to O in mode A is removed from the current access matrix.

11.8 Create_Object.

Create_Object (S, O1, L1, O2) means that S creates O1 at level L1 and O2 is the Parent of O1.

This request is granted if the following is true:

S has either append or write access to O2

AND

L1 dominates the level of O2.

If the request is granted then O1 is a new object with Parent O1 and the level of O1 is L1.

Evidently ROOT exists in the initial state.

11.9 Delete_Object.

Delete_Object (S, O) means that S deletes O and all objects inferior to it.

This request is granted if the following is true:

O is not ROOT and S has write access to the parent of O

If the request is granted then all current accesses to O and all objects inferior to O are removed from the current access matrix and all permissions for access to O and objects inferior to O are removed from the permission matrix.

11.10 Change_Subject_Current_Level.

Change_Subject_Current_Level(S, L) means that the new current level of S is L.

This request is granted if the following is true:

the maximum level of S dominates L AND either S is trusted or all current accesses by S obey the star-property assuming that the current level of S is L

If the request is granted then the new current level of S is L.

11.11 Change_Object_Level.

Change_Object_Level(S, O, L) means that S changes the level of O to L.

This request is granted if the following is true:

S is trusted and the maximal level of S dominates the level of O or the current level of S dominates L and L dominates the level of O

AND

if any subject T has access to O in read or write mode then the current level of T dominates L

if O's level were changed to L then the star-property still holds for all current accesses to O

AND

if O's level were changed to L then the compatibility property would still hold in the object hierarchy

AND

CHANGE(S, O) is true, which means that S may change the level of O. CHANGE is not defined further in the report.

If the request is granted then the new level of O is L.

12 SOME CRITICISMS OF THE MODEL

Now that we are familiar with the model, I would like to indicate some of the problems with it.

In general, the notation is hard to follow. In particular, the explanation of the underlying state machine and the metatheorems concerning security-preserving rules, on pages 87 to 99, is unnecessarily complicated.

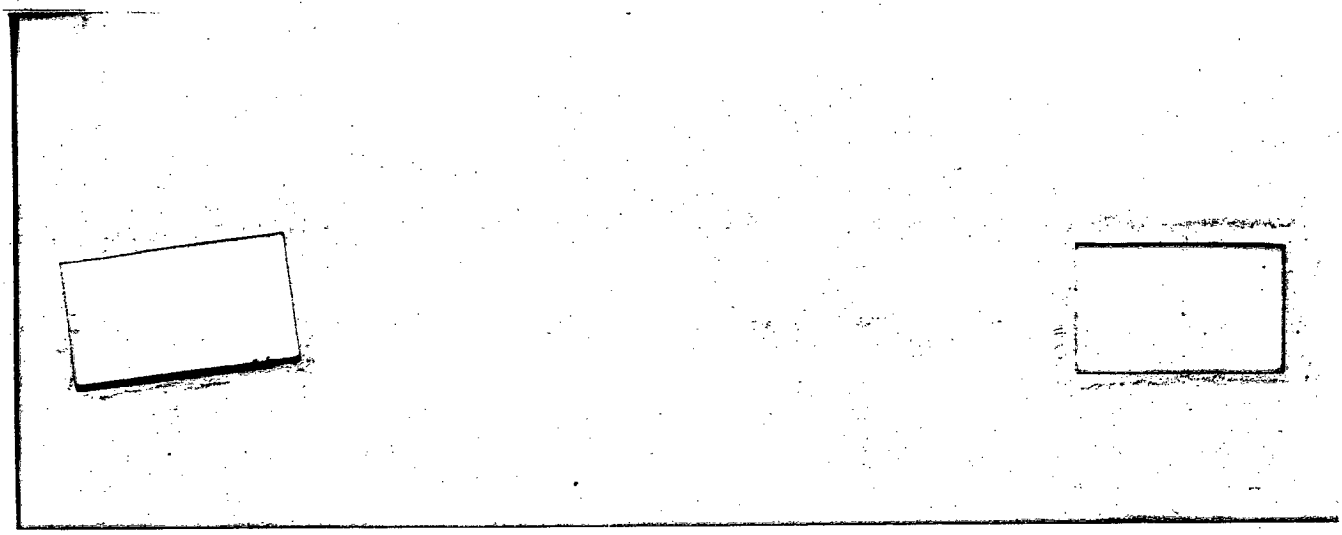
It should have been pointed out somewhere that "simple security" is implied by the "star property" unless the access is being requested by a trusted subject. Otherwise, one is tempted to think the use of maximal level of a subject in some places and the use of current level of a subject in others are typographical errors.

The informal statement of Rule 6 and its formal statement differ. The rules for Give_Permission (rule 6) and Rescind_Permission (rule 7) are incomplete because the functions GIVE and RESCIND are not completely defined--indeed they are not mentioned elsewhere in the report! In fact, the security policy has no requirements regarding modification of the permissions matrix. This seems to me to be a poor choice for later refinement. Rules for altering the permission matrix are at least as important as the rules for writing (other) objects.

CHANGE, in Rule 11, is also not defined formally.

The rules for an object hierarchy imply an awkwardness in creating and editing files. It turns out that if one is working in a multi-level directory, then one must change one's level in between creating a file and editing it. This is clearly not desirable in a real system. This particular feature is presented in detail in the next section.

Including Change_Subject_Level and Change_Object_Level transformations puts an extra burden on the implementation of a secure system. To be specific, the system must be capable of tagging individual process memory buffers with security labels and checking for improper access if the process ever changes its label. If this is not done, then it is easy to do an unauthorized "write down." Just read a file at one level into a program buffer, change level, and write the buffer into a file at a lower level.



13 THE PROBLEM WITH HIERARCHIES

This section discusses a problem that arises in working in a typical file hierarchy if the access rules of the B-L model are strictly adhered to. This discussion concerns UNTRUSTED users (subjects).

I are interested in what the B-L model would imply, when a user creates a file in a directory and proceeds to edit that file. I contend that the model's rules imply

EITHER

the user must change (actually "raise") his/her current security level after creating the file and before opening the file for read-and-write access;

OR

the directory and the new file and the user must all be at the same security level.

Thus an inconvenient and (possibly insecure) change-level operation must frequently take place, or painless text processing can occur only in single-level subtrees.

A real system is likely to adhere to a tranquility principle for untrusted subjects though trusted subjects would be able to change their level and the level of objects. Such a system cannot follow the security requirements of the B-L model and be user friendly.

Let's now consider the scenario in detail and show that the problem really does exist.

Suppose a user U decides to create a new file F in a already existing directory D. The request takes the form, Create (U, F, D, L), where L is the proposed security level of F.

The applicable rule is #8 Create_Object, p. 118.

There are two conditions that must be true in order for this request to be granted.

1. U must have access to D in either read-and-write or write mode

2. L >= level (D) (compatibility requirement)

In order for U to have write access to D, it must be true by the star-property (p. 86) that

level (U) <= level (D)

Thus, at the time F is created, we have

level (U) <= level (D) <= L

Assume now that D decides to edit F WITHOUT CHANGING ANY SECURITY LEVELS. That is, D proceeds to immediately access F in read-and-write mode.

The applicable rule is #4 (Get_Write), p. 115.

In order for this request to go through, it must be that

level (U) = level (F)

But,

level (F) = L.

Combining this with the previous inequality we see that

level (U) = level (D) = level (F)

Thus, if creating and editing of files is to be painless, all users and files must have the same level!

What ever happened to multi-level security? The only way out of the bind just presented is to relax the requirement for writing directories. The other alternatives--allowing low level subjects to browse high level directories or allowing untrusted subjects to write down--are not acceptable.

In the model for a Unix file system which I will describe next, I will present an interpretation of directories that will make a relaxation of the star property seem reasonable.

B1 - LEVEL SECURE UNIX FILE SYSTEM

CONTENTS

A SECURE FILE SYSTEM FOR A B1-LEVEL UNIX

1 INTRODUCTION

The definition of B1-Level security is contained in the United States Department of Defense publication, "Department of Defense Trusted Computer System Evaluation Criteria" (the Orange Book). It is widely assumed that these criteria for certification will increasingly become a standard for computer systems that handle classified data or that perform sensitive tasks within the U. S. Government. Since Unix is a popular operating system with the government, there is a great deal of effort going into producing certifiably secure Unix systems.

This model for the Unix file system is one subset of the software design decisions that could be imposed on a standard Unix to yield a B1-Level secure system. Nothing will be said about implementation choices. It will be obvious in some places that, although software is being discussed, hardware and firmware design is addressed implicitly.

I would like to emphasize at the outset that nothing in the realm of formal methods applicable to producing secure systems cannot also be applied to producing systems that must meet other requirements such as very high reliability or, at a smaller granularity of concern, absence of deadlock.

It's silly to identify formal techniques of specification and verification with a single application of those techniques.

Before describing the model for file system access I will cover some background material.

2 SECURITY LABELS (LEVELS)

In a secure system, users and their processes and objects such as files and devices have security labels associated with them. It doesn't matter for our discussion what these labels look like or how they are compared as long as we know that it is always possible to decide of two given labels, L1 and L2, that either L1 = L2, L1 <= L2, or that L1 and L2 are not comparable. Inevitably, mandatory checks require either that L1 dominates L2 or that L1 equals L2. The check always fails if the labels are not comparable.

In this talk, "security level" and "security label" or just "level" or "label" are used interchangeably.

An important point is that in this talk the system is assumed to obey a "tranquility principle." That is, once a user or an object comes into being with a level attached to it, that level never changes. In practice, that means that to change levels, a user must logout and login again. Each user has a maximum level at which he/she may operate, and when logging in a level dominated by the assigned maximum is selected for that session. A file must be copied or renamed in order to change classification. This assumption greatly simplifies analysis of a security policy.

3 USERS: PRIVILEGED AND ORDINARY

In standard Unix there is just one privileged user, the "root", that has ALL privileges. Other users are constrained by discretionary access controls and specific checks for the root id. Unix operates in two modes, single user and multi-user. In single user mode, the "root" is the only user.

In a secure Unix, there would be still be the distinction between single-user and multi-user mode. Only in single user mode, would a "root" user be allowed. Unless stated otherwise, we are always describing behavior when the system is in multi-user mode. In a secure system, there would be different categories of privileged users, none of which would have all of the privileges of a root user. In the rules given later, exceptions for privileged users are always stated explicitly. For simplicity however, we will not distinquish among categories of privileged users. In general, privileged users may ignore discretionary access controls, and mandatory controls on security level will often be relaxed to refer to the maximum level of a privileged user instead of the current level.

New processes spawned from old processes always have a subset of the privileges of the parent. In particular, the security level of the child process is the same as that of the parent. This implies that the set-user-id and set-group-id modes of an executable file that enable a process to assume a new id and group while executing the file are not present in our secure Unix.

Privileged processes may change their levels and the levels of objects.

4 Bl-LEVEL SECURITY

My main discussion--file system security--is only a part of security as defined by the Orange Book. I will review briefly the total security picture as the Orange Book sees it. In general, these aspects of a secure system will not be discussed in succeeding sections.

Following the Orange Book, security is divided into the following areas

1. Security Policy

A security policy is a set of requirements that security-related parts of the system must meet. Put a little differently, a security policy is a statement of how the system will behave in matters that are relevant to the prevention of an undesirable flow of information. This flow is usually visualized as an undesirable copying of data from a highly classified file to a file of lower classification (as a first step towards a more sinister migration).

However, "security" is more than the leaking of classified data. Security of a system involves correct operation of its parts and the protection of those parts from corruption, either deliberate or accidental.

A policy must address discretionary and mandatory control mechanisms, object reuse, and security labeling of system objects.

Object reuse requirements guarantee that when a system resource, a disk sector, say, is reused, the data previously contained is deleted.

Discretionary and mandatory control mechanisms for file systems and the labeling of files will be discussed later on. However, I won't discuss labeling other objects such as memory segments, devices, and human-readable output.

Discretionary control rules are rules whereby a user may at his/her own discretion allow other users access to certain system objects under his/her own control.

Mandatory controls, however, are controls that the operating system enforces for all users and all objects. Exceptions must be written into the rules themselves. A specific mandatory policy is introduced at the B1 level. Put simply, it says that a user at level A may read an object at level B only if level A dominates level B, and a user may write the object only if level B dominates level A. These two requirements are frequently refered to as "simple security" and "star property", respectively. The second is also refered to as "no write down." As we shall see, this policy causes some problems in dealing with directories.

It is important to note that access to an object is granted only if BOTH discretionary checks are passes and mandatory checks are passed.

2. Accountability

This area includes user identification and auditing. Secure systems will probably include, in addition to what the Orange Book mandates, the features described in a document published just this year, "Department of Defense Password Management Guideline". It is likely that this guideline will become a satellite standard.

A superior B1 system would include the B2-level requirement of a "Trusted Path" between the user terminal and the operating system for use during login.

3. Assurance

System architecture requirements fall into this area. A recent hardware evaluation guideline issued by the National Computer Security Center (NCSC) mandates that at even at the C2 level the hardware must enforce at least two domains of main memory, system and user; and process isolation between users and between all users and the system.

A superior B1 system would also include the B2-level requirement of separation (in software) of operator and administrator privileges.

Another requirement in this area is a formal or informal model of a security policy. This talk is an effort to define such a policy. A formal policy is required for a B2 system.

Requirements for a test suite fall within this area. I won't say anything about testing except that testing and documentation together account for about half of the effort of a Unix system upgrade to certifiability.

4. Documentation

This talk doesn't address documentation requirements.

5 DISCRETIONARY ACCESS CONTROL

Though not required by the Orange Book, it's likely that every B1 system will have access control lists. For our purposes we lose no generality by assuming that the discretionary access control mechanism is the standard Unix owner-group-public mode bits.

Each file and each process has associated with it a group id. Processes may belong to any of several authorized groups but at any one time a process belongs to a single group. There is a command to change from one group to another one. Each file in Unix has a mode word that contains nine bits--three each for owner privileges, group privileges, and public privileges. For example, if the mode word is displayed as

```
-rwx-r-x-r--
```

Then the owner of the file may read, write, and execute the file. Someone not the owner but a member of the same group as that of the file may read or execute the file. Someone not the owner nor a member of the group of the file may only read the file. A privileged user, however, may exercise read, write, or execute access regardless of the mode settings.

The owner of a file is the user that created it. Only the owner or a privileged user may change the access modes of a file.

6 LINKS, HARD AND SOFT

A hard link to an already existing file is simply another path to that file that cannot be distinquished from the file itself. Creating a file is simply establishing the first link. As far as the security policy goes, the requirements for making a link will be the same as those for creating the file if this is the first link, or at least as strict as copying the file if the link is to an already existing file.

The idea is that if security can be violated with a link, then it can be violated without the link.

Symbolic (soft) links, on the other hand, can be distinquished as links and exist as separate files which contain a pathname to the file being linked to. Symbolic links exist so that links into different mountable file systems can be made. This implies that a symbolic link to a file that doesn't exist or is not now visible is permitted. Creation of a symbolic link is subject to the same checks as creation of an ordinary file.

Access to a file through a symbolic link is permitted only if the link is accessable and the file pointed to is accessable. For example, if A is a symbolic link to B, then a user U can read B only if U can read the file A and can read the file B. A file cannot be deleted using only a symbolic link to it. Only the symbolic link is deleted.

7 DIRECTORIES AS CONTAINERS

Directories are often thought of as containers of files or as containers of file names. Either way, the star property makes directories as data files difficult to deal with.

The problem is easily stated. To "read" a directory or have "execute" access to it, a user's level must dominate that of the directory. To create a file in that directory, the user must have "write" access to the directory thus the level of the directory must dominate the level of the user. These two things require the level of the user and that of the directory to be the same. On the other hand, if the user, after creating the file now attempts to edit it; that is get simultaneous read and write access to the file, then the level of the user must be equal to the level of the file. Thus all levels, user, directory, and file are the same!

What ever happened to multi-level security?

The way out of this is to consider directories as corridors instead of as containers. Thus the check for write access can be relaxed. We will think of directories as corridors providing access to file names, but just as you may walk into a corridor and come to a door you can't open, so you may read a directory but not be able to see a particular file name. Reading a directory will not be an all-or-none operation. Each file name in the directory has a security level. The level of a file name is the same as that of the file. A file name will be displayed only if the user has read access to the directory and has read access to the file.

In this scheme, as far as mandatory checks go, if you can read a directory, you can write it, and a directory may contain file names at a higher level that that of the directory. Only a privileged user may place a name in a directory where the level of the name is lower than the level of the directory.

A similar problem of interpretation arises with execute access when considering a pathname composed of two or more directories. Suppose a user requests execute access to /A/B. The user must first be given access to A and then, in a separate check, be given access to B. This shouldn't be a surprise. It is exactly the way discretionary access works.

8 SINGLE-LEVEL DIRECTORIES

Each directory is either a "multi-level" directory or a "single-level" directory. A multi-level directory may include file names at levels different from its own level. A single level directory includes only names at the same security level as the directory itself. This implies that all files in the subtree beneath a single-level directory are at the same level, and that all directories below a single-level directory are single-level directories at that level. Not even privileged users may place a file in a single-level directory at a level different from the directory.

Single level directories give administrators the option to separate files according to security levels. Using single level directories avoids obvious covert channels which exploit the collision of names at different levels and avoids the nuisance of not being able to see all of the files in a directory at one time.

9 FILE SYSTEMS

This section defines a security policy for file systems in a Bl-Level Secure Unix.

9.1 Directories.

9.1.1 Access Modes.

In this section we review what the three access modes, read, write, and execute mean when applied to a directory.

To read a directory is to read the names of the files for which the directory provides direct access. Only the names of files that the user has read access to can be read.

To write a directory is to change the list of names in the directory. Thus, creating and deleting files constitutes writing the directory that contains the names of those files.

As we shall see, deleting a file requires write access to the file as well as the directory.

Execute access to a directory simply means that further access, of any kind, to files in the subtree under the directory is now possible but not automatic--actual access being subject to checks at the file level. This is sometimes described as a "search" property. If one does not have execute access to a directory then one cannot read, write, execute, create, or delete any file whose name is in that directory.

9.1.2 Mandatory Policy (Rules About Labels).

In the rules that follow, U denotes any user, privileged or not. A lesser requirement applicable if a user is privileged is always explicitly stated. D denotes a directory. F denotes either a directory or an ordinary file unless specifically described as a link. L denotes a label (level), and L(U), say, denotes, the current label (level) of U.

In order for any access to be possible to a file, the user must have execute access to all directories superior to the file.

execute

If U gains execute access to D, then L(U) >= L(D).

If U is privileged, then the maximum level of U must dominate L(D).

read

If U gains read access to D, then U must already have execute access to D.

If the name of a file F, or a hard link to F is displayed during a read access, then L(U) >= L(F).

If U is privileged, then the maximum level of U must dominate L(F).

If F is a symbolic link, then the name is displayed only if the level of U dominates the level of the link itself--NOT that of the file actually pointed to by the link.

If U is privileged, then the maximum level of U must dominate the level of the link.

write

If U gains write access to D, then U must already have execute access to D.

If D is a single-level directory, then L(U) = L(D).

If U creates or deletes a file F or a hard link to an already existing file F, then L(U) = L(F).

If U deletes a file F or a hard link to F, then U must already have discretionary access to F in write mode.

If F is a symbolic link (and thus a file

separate from the file being linked) and is being created or deleted, then L(U) = L(F). A symbolic link is created at a level of the user not at the level of the file being linked. If a symbolic link is deleted, then only the link itself is deleted. The file that was linked, if it exists, is untouched.

If U is privileged and U is creating or deleting any kind of file F, then only the maximum level of U need dominate the level of F. However, not even a privileged user may create a file in a single-level directory if the level of the file is not that of the directory. Also, directories under a single-level directory are all single-level.

9.1.3 Discretionary Policy (Rules About Modes).

The discretionary checks are those described earlier in the section on Discretionary Access Control. Privileged users are not subject to discretionary access checks.

9.2 Ordinary Files And Links.

9.2.1 Access Modes.

If a file is designated as an executable file by the system, then execute access means that the file or the file being linked may be executed by the user.

The meaning of read access to an ordinary file or to a file through a link is what the name suggests--the way is clear for the contents of the file to be copied into the address space of the user.

Write access to an ordinary file or to a file through a link means that the way is clear for the user to modify the contents of the file.

However, in all the above cases, access to a file through a symbolic link is possible only if access is granted to the link and to the file being linked.

If the user also has write access to the directory containing the name of the file, then the user may delete the file.

If there are multiple hard links, only the specified link is deleted.

If a delete request is made using a symbolic link then only the specified symbolic link is removed.

9.2.2 Mandatory Policy (Rules About Labels).

In the rules that follow, U denotes any user, privileged or not. A lesser requirement applicable if a user is privileged is always explicitly stated. F denotes either an ordinary file or a link. L denotes a label (level), and L(U), say, denotes, the current label (level) of U.

In order for any access to be possible to a file, the user must have execute access to all directories superior to the file.

execute If U has execute access to F then L(U) >= L(F). If F is a symbolic link then the level of U must also dominate the level of the file that F points to, and U must have execute access to all directories superior to the file being pointed to.

If U is privileged, then only the maximum level of U need dominate L(F). If F is a symbolic link, then only the maximum level of U need dominate the level of the link and the level of the file being pointed to.

read If U has read access to F then L(U) >= L(F). If F is a symbolic link then the level of U must also dominate the level of the file that F points to, and U must have execute access to all directories superior to the file being pointed to.

If U is privileged, then only the maximum level of U need dominate L(F). If F is a symbolic link, then only the maximum level of U need dominate the level of the link and the level of the file being pointed to.

write If U has write access to F then L(U) = L(F). If F is a symbolic link then the level of U must also equal the level of the file that F points to, and U must have execute access to all directories superior to the file being pointed to.

If U is privileged, then only the maximum level of

U need dominate L(F). If F is a symbolic link, then only the maximum level of U need dominate the level of the link and the level of the file being pointed to.

Deletion and creation of ordinary files and links is covered in the section on mandatory policy for directories.

9.2.3 Discretionary Policy (Rules About Modes).

The discretionary access rules for ordinary files and links were described in the section on Discretionary Access Control except that where symbolic links are concerned, access must be granted to both the link and to the file being linked.

Privileged users are not subject to discretionary access checks.

9.3 Root-Owned Files.

To enhance the integrity of system executable files, libraries, and other system databases, an additional mandatory policy is enforced regarding any file whose owner id is 0--the same as the "root" id. Namely, no user who is not privileged may write or delete a root-owned file, whether directory, link or ordinary file.

Privileged users may write root-owned files if allowed to by the mandatory checks previously described.

9.4 Device Special Files.

In Unix, input and output to devices is handled as reads and writes to special files. Read and write access to special files involving labels and file modes is the same as for non-executable ordinary files except for a subset of "protected" devices which includes tape drives, and user terminals. For these device files, there is a mandatory access control mechanism that replaces the discretionary and mandatory controls already described. This mechanism governs both privileged and non-privileged users.

The system maintains a table of owner ids for all protected devices. An owner or a privileged user may have any access to the device. A user not privileged and not the owner of a device is denied all access to that device. This mechanism makes it difficult for one user to interfere with another user by reading or writing the other user's terminal. It also allows controlled access to tape drives by trusted servers.

10 SUGGESTIONS FOR FILE SETUP

Here are some suggested settings for levels and modes for typical files in a Unix system.

1. Slash and other system files such as bin/, etc/ should be owned by root with group staff. Discretionary modes should be rwxr-xr-x. Level should be "system-low." These same settings are suitable for system binaries and library files. Being owned by "root," these files are safe from modification by non-privileged users. Being classified as system-low, they can be used by everyone.

2. In general, the security level of files increases as one goes further now in the file hierarchy.

3. Single-level directories can be used to encompass work on a specific project where all files can be at the same security level. Discretionary controls can keep other users out.

4. Users in some work spaces can be further isolated from the rest of the system by establishing a "gateway" directory above their route directory.

   For example, suppose A is a single-level directory at level L that is to be isolated from the rest of the system. A could be placed by a privileged user under a directory Z which is at level "system-high". Only privileged users can operate at level, system-high. Users upon logging in would be tagged at level L and placed inside A. Other non-privileged users do not have execute access to Z and cannot access any files beneath Z. The separation is not complete and is not meant to be. Users inside Z can, for instance, access system executable files by using full pathnames instead of relative pathnames going through Z.

   There are stronger methods for isolating a file system if one wishes to have complete separation.