A Multilevel Security Model For Distributed Object Systems

Vincent Nicomette and Yves Deswarte

LAAS-CNRS & INRIA
7, Avenue du Colonel Roche
31077 Toulouse Cedex - France
Telephone: +33/61 33 62 88 - Fax: +33/61 33 64 11
(nicomett@laas.fr, deswarte@laas.fr)

Abstract. In this paper, the Bell-LaPadula model for multilevel secure computer systems is discussed. We describe the principles of this model and we try to show some of its limits. Then we present some possible extensions of this model, with their drawbacks and advantages. We finally present our own extension of the model for object-oriented systems. In this last section, we first explain the principles of our security policy, then we describe the rules of our authorization scheme and we give an example of a typical scenario in a distributed object-oriented system.

1 Introduction

During the last few years, object-oriented programming has been the most important issue of software engineering. Thanks to its power of abstraction and re-usability, the object model appears to be suitable to the development of nowadays systems which are increasingly complex. But this object model does not fit easily the usual protection models. As a matter of fact, most of these protection models are based on the notions of subjects and objects that do not correspond to the notion of object as defined in the object-oriented languages. Furthermore, the information flows in object systems have a very concrete and natural embodiment in the form of requests and corresponding replies. To prevent confidential information leakage through these information flows, we propose a multilevel security model.

As such, our model is similar to the well-known Bell-LaPadula model, which is promoted by the US Department of Defense. Bell-LaPadula 's rules do prevent illegal information flows but are often too restrictive. The purpose of this paper is thus to present a confidentiality model which has the same properties as the Bell-LaPadula model but which is less restrictive. Section 2 is dedicated to the presentation of the Bell-LaPadula model and its main drawbacks. Section 3 presents some extensions which have been proposed for this model. Section 4 introduces our protection model and gives an illustration of this model by means of a detailed example. Section 5 demonstrates how this model prevents illegal information flows. Section 6 compares this model to existing models and section 7 presents some perspectives for future work.

2 The Bell-LaPadula model

The Bell-LaPadula model [1] is a prominent model for mandatory access-control mechanisms enforcing a multilevel security policy in automated information-processing systems. The Bell-LaPadula model was adopted as the formal policy in the US Department of Defense's influential Trusted Computer System Evaluation Criteria (TCSEC), commonly known as the Orange Book [2]. In order to obtain the "B" classification of the Orange Book, a system must enforce a mandatory access-control mechanism based on the Bell-LaPadula model's formal security policy.

The Bell-LaPadula model is a state machine model that abstractly describes a system in terms of system *states*, and *rules* that enable transitions between states. The state of the system includes a set of triples that define the current access mode each *subject* has to each *object* in the system. A subject represents an active entity, such as a process. An object is a passive container of data, such as a file. The set of all modes in which a subject can access an object is (*execute*, *read*, *append*, *write*):

- *execute* corresponds to: neither observation nor alteration;
- *read* corresponds to: observation with no alteration;
- *append* corresponds to: alteration with no observation;
- *write* corresponds to: both observation and alteration.

Different security levels are defined and partially ordered. Each object is assigned one security level which is called the *classification* attribute of the object and each subject is assigned two security levels: the first one represents the *clearance* of the subject and is a static level, the second one is the *current security level* of the subject; it is required that the clearance of the subject dominates the current security level of the subject.

This model describes two access-control rules:

- **The simple security property:** A state is secure iff, for every subject in the system with the ability to observe an object, the subject's clearance is greater than or equal to the object's classification.
- **The *-property:** A state is secure iff no subject may observe the contents of an object O_1 and store information in an object O_2 unless O_2's classification is greater than or equal to O_1's classification.
  The definition of the *-property can be refined in terms of current security level:
  A state is secure iff for each access (subject, object, access mode) in the system :
  - the object classification dominates the subject current security level if the access mode is append;
  - the object classification equals the subject current security level if the access mode is write;
  - the object classification is dominated by the subject current security level if the access mode is read.

The simple security property prevents users from accessing information they are not cleared to access and the *-property prevents the information flow from a high level of classification to a lower level of classification in the system.

The main drawback of this model is that the *-property is too restrictive. For example, in a system which enforces a security policy based on the Bell-LaPadula model, let us consider a user, logged in as SECRET, who reads a CONFIDENTIAL file and then makes a copy of this file. The *-property requires the user to create the copy with a classification which is at least SECRET. But obviously the classification of the information which is contained in the copy is the same as the classification of the information of the original file, i.e., CONFIDENTIAL. Thus, the *-property requires the user to create a file with a label that does not correspond to the real classification of its content. This example puts the emphasis on one of the main drawback of the Bell-LaPadula model: the classification of the information in the system goes on increasing. This increasing leads to a degradation of the information in terms of accessibility. Thus, such a system needs the intervention of trusted subjects in order to periodically declassify the information. *Trusted subjects* are subjects that can be relied on not to compromise security. Thus their operations in the system are not submitted to the access control rules. They can declassify information, which is an illegal operation according to the access control rules of the model. In sum, the main problems with this model are not the things it allows but the things it disallows [3]. Many operations that are in fact secure will be disallowed by the model. The systems that choose to use the Bell-LaPadula model either strictly implement it and thus accept a degradation in the functionality of the system, or implement a lot of services as trusted subjects.

A second important drawback of the Bell-LaPadula model has been underlined in many papers: this model does not prevent the existence of covert channels. *Covert channels* are paths not normally intended for information transfer at all, but which could be used to signal some information towards lower levels. The problem of covert channels is directly connected to the sharing of different resources of the system (storage resources or timing resources). A covert channel may be used in two ways:

- directly by a malicious user with a high clearance who intentionally transmits information to a malicious user with a low clearance; in that case, the users both collaborate in order to realize illegal information flows;
- indirectly by a Trojan horse: a user with a high clearance involuntarily executes the Trojan horse which transmits information to the malicious user with a low clearance.

As originally formulated, some of the rules in Bell-LaPadula model allowed information to be transmitted improperly through control variables (storage channels). As reported by Landwehr in [3], Walter et al. established a final form of the model in which the rules of the model do not contain storage channels, but in which timing channels can exist. Unfortunately, timing channels can be implemented easily, if a resource is shared between high and low users. For example, a process p_1 might vary its paging rate in response to some sensitive data it observes. Another process p_2 (whose current security level is inferior to the current security level of p_1) may observe the variations in paging rate and "decipher" them to reveal the sensitive data. There is no rule in the Bell-LaPadula model that can prevent such an information flow. Further reading about the Bell-LaPadula model and its drawbacks can be found in [4], [5], [6].

3 Some extensions of the model

3.1 The floating labels method

In [7], John Woodward explains that the over-classification problem stems from the fact that, in traditional implementations of secure/trusted systems, such as SCOMP [8], subjects and objects inherit the sensitivity label of their creator. If we consider the example given in the previous section, when the user logs in at a SECRET level, the system creates a SECRET command interpreter to service him. When the user enters a copy file command, the command interpreter creates a SECRET process to run the command. This process is SECRET because it is created by a SECRET level command interpreter. The copy process creates a new file into which it intends to copy the CONFIDENTIAL file but this new file must be created SECRET thus over-classifying the data.

John Woodward proposes to associate two labels to each process and data in the system. The first label (called the sensitivity label) is a floating label that is intended to represent the *actual* sensitivity of the data stored in an object. Upon creation, each object must have the lowest sensitivity level of the system because it contains no data. In the same way, upon subject creation, the sensitivity label of the subject must represent the sensitivity of the data of its address space. Furthermore, if a subject executes a system call in such a way that its whole address space is reinitialized, then the sensitivity label of the subject must represent the sensitivity of this new address space even if this new sensitivity label is lower than the previous one. The second label that is associated with each object and subject of the system is a security label exclusively used for mandatory access control. This label is called MACL (Mandatory Access Control Level). It represents for an object the classification of the data that it contains and for a subject the clearance, i.e., the maximum level of data that it can read. The sensitivity level of a subject or object can never exceed its MACL. An example of such a labeling is detailed in Figure 1.

J. Woodward argues that this method allows to properly label a file which is to be exported from the system because the sensitivity label represents the actual level of information stored in the file. He also explains that any user in the system can set (using a trusted/privileged system command) a file's security level to the sensitivity level of this file when he wants to share this file with someone he trusts.

It seems to be interesting to keep this sensitivity label with each object and subject in the system. As a matter of fact, if we consider the declassification process, the sensitivity label allows to declassify the information at a level that

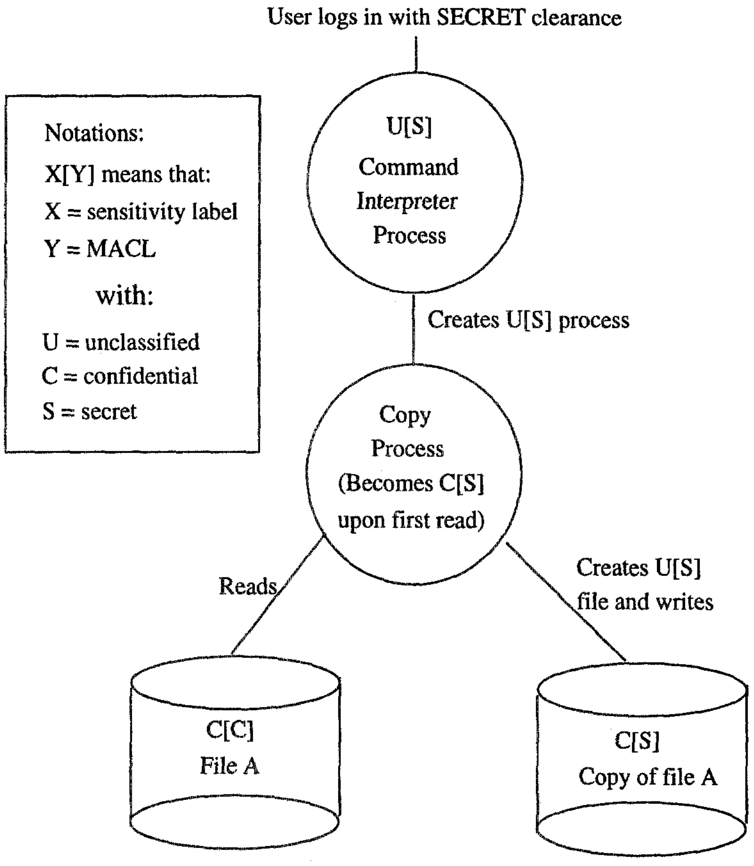

Fig. 1. Floating labels

is really the actual level of the information. Thus, there is no need for trusted users in order to estimate the information which is to be declassified.

The main drawback of this model is the creation of covert channels that do not exist in the Bell-LaPadula model, in particular storage channels. For example, let us imagine a user p with a SECRET clearance who wants to give information to a user q with a CONFIDENTIAL clearance. The user p declassifies several files from SECRET classification to CONFIDENTIAL classification (in order to be authorized to realize this declassification, the user p had taken care of not writing information with classification higher than CONFIDENTIAL in these files). When the declassification process is realized, the user q can read all these files and in particular their names. If the user q knows that a name which begins with a vowel means 0 and that a name which begins with a consonant means 1, there is an illegal information flow thanks to this covert channel.

There are also timing channels in this model. Let us imagine for example a user p who chooses to declassify some files at some very precise dates. If a user

q with a lower clearance can notice these precise dates (each date corresponds to the precise moment when he can read one more file), then he can use these dates to receive information from p.

3.2 Causal dependencies

Bruno d'Ausbourg [9] presents an original way to model the information flow in multilevel systems. He describes a system as a set of points. A point *(o,t)* references an object o at a time t. These points evolve with time and this evolving is due to elementary transitions made by the system. An elementary transition can modify a point: then, at instant t, it sets a new value v for the object o of the point. This instant t and the new value v functionally depend on previous points. This dependency on previous points is named *causal* dependency. The causal dependency of *(o,t)* on *(o',t')* with $t' < t$ is denoted by $(o',t') \rightarrow (o,t)$. The *causality cone* is defined as:

$cone(o,t) = \{(o',t')/(o',t') \rightarrow^* (o,t)\}$

where $\rightarrow^*$ is the transitive closure of the relation $\rightarrow$.

These causal dependencies makes up the structure of information flows inside the system. If a subject has any knowledge about the internal functioning of the system, then he is able to know the internal scheme of causal dependencies. So if a user has knowledge of the internal functioning of the system and if he can observe an output point x_0, then he is able to infer any information in $cone(x_0)$, i.e., he is able to observe all the points in $cone(x_0)$.

With respect to disclosure of information, a system is considered secure if a subject can observe only the objects he has the right to observe. If we note Obs_s the set of objects that a user s can observe and R_s the set of objects that this user is authorized to observe, we can say that the system is secure if: $Obs_s \subseteq R_s$.

In a system enforcing a multilevel security policy, d'Ausbourg explains that two conditions are sufficient to guarantee the security of the system. In such a system, a classification level *l(x)* is assigned to points x and a clearance *l(s)* is assigned to subjects s. The first condition is that the clearance of a user s dominates the set of output points O_s that he can observe:

$\forall s, x_0 \in O_s \Rightarrow l(x_0) \leq l(s)$

The second condition requires a monotonic increasing of levels over causal dependencies:

$\forall x, \forall y, x \rightarrow y \Rightarrow l(x) \leq l(y)$.

This model is interesting because it contributes in a new way of formalizing the information flows in a system. Furthermore, what makes this formalization interesting is its minimal aspect: the notion of causal dependencies allows the information flows to be described in a minimal way. Thus, if we define a security policy and an authorization scheme based on this notion of causal dependencies, the rules of the authorization scheme can describe the minimum conditions that are to be enforced in a system in order to prevent the information flows. From this point of view, this model is better that the Bell-LaPadula model that controls the information flows with too severe measures. For example, in the model based on causal dependencies, a user SECRET can change the label of an object from

SECRET to UNCLASSIFIED if it is reinitialized. As a matter of fact, the dependencies that can be established for this operation do not lead to illegal information flows.

The main drawback of such a model seems to be the difficulty to enforce it in real systems. As a matter of fact, it seems to be difficult to very closely establish the causal dependencies in a system, just as the set of objects that may be observed by a subject. Furthermore, the author does not indicate any formal method which could help the administrators to precisely evaluate the causal dependencies in a system. And this evaluation even becomes more tricky in distributed systems where subjects and objects from multiple sites may co-operate.

4 A multilevel security model for distributed object systems

4.1 Secure entities

Our model is based on two main concepts: the notions of *objects* and *activities*. These entities are assigned labels as will be explained in the next paragraph.

The objects of our model are objects as defined in the object-oriented languages. Each object is made up of a private state information and a set of operations which represent the object interface. The operations defined on an object are called *methods* and are the only way to modify the state of the object or to get information from this state. The communication between objects consists in sending messages. An object O calls a method of an other object O' by sending O' a request. This message consists of a method selector and a list of arguments.

An activity in distributed object systems consists in a succession of method executions and requests through different objects of the system. This set of method executions and requests are dependent and collaborate to achieve a high-level task (e.g., printing a file on a printer). An activity exchanges information with the different objects it accesses. This notion is similar to the notion presented for the Chorus micro-kernel in [10]. In Figure 2, an activity is represented. This activity realizes a high-level task: recording a scene. This activity consists in:

1. Starting the execution of the method *Record-Scene* of the object *Client*,
2. Making a request to the method *Record* of the object *Recorder*,
3. Starting the execution of the method *Record* of the object *Recorder*,
4. Making a request to the method *Take* of the object *Movie-Camera*,
5. Executing the method *Take* of the object *Movie-Camera*,
6. Returning to the method *Record*,
7. Making a request to the method *Write* of the object *Video-Tape*,
8. Executing the method *Write* of the object *Video-Tape*,
9. Returning to the method *Record*.
10. Returning to the method *Record-Scene*.

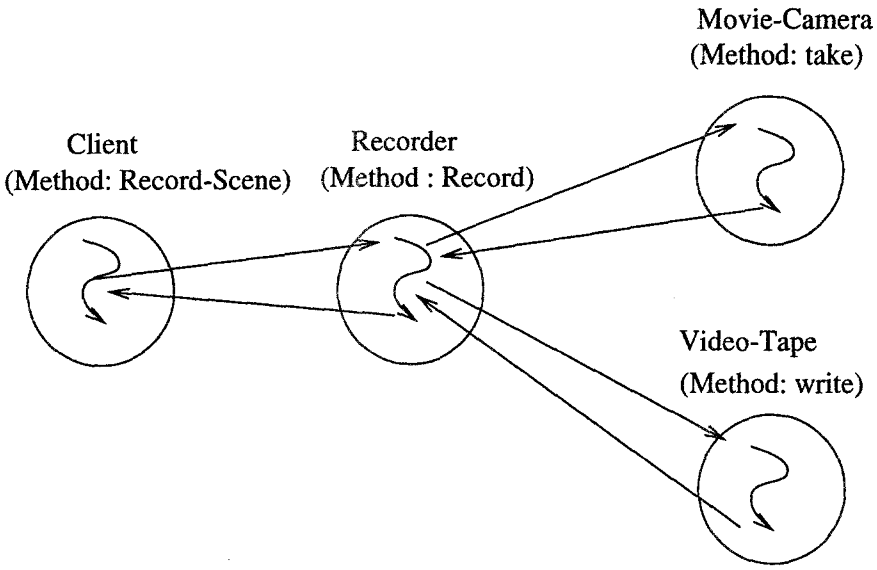

Fig. 2. An activity

The point in the notion of activity is the idea of dependence, of collaboration: an activity is made of a set of method executions that form parts of the same global operation from a user point of view.

One of the important features of our model is that we distinguish two object families: the *stateless objects* and the *stateful objects.* In the previous example, the activity flows information from the object *Movie-Camera* to the object *Video-Tape.* The object *Recorder* does not keep any memory of the information that it receives from the object *Movie-Camera*; it simply writes this information in the object *Video-Tape.* Yet, the object *Recorder* is accessed by the activity as the two other objects. We can generalize this idea: any object stores data but these data may be classified in two categories: the system data and the application data. The application data are the data which compose the state of the object from the application point of view, i.e., the data that are declared in the source code of the object by the programmer. The system data are data added by the system in the object to control the execution of the object (for example the process stack). A stateless object is an object which does not store application data: its state is initialized at each invocation. There is no information flow between two successive requests that access a stateless object. This notion of stateless object is analogous to the notion of "object reuse" of the Orange Book [2]. Conversely, a stateful object is an object which stores application data and whose methods consist in reading or writing these data. A lot of examples of stateless objects can be found in classical systems such as Unix. For instance, an NFS server is a stateless object. This server does not keep in memory any of the requests that it receives. Its state is initialized at each invocation. All object managers, in general, are stateless objects (browsers for example). We will see in the next section that this notion of stateless object will be useful to implement a less restrictive authorization policy than the Bell-LaPadula policy while preventing in the same way the illegal information flows.

The goal of a multilevel security policy is to prevent users to get information they are not cleared to access. As we suppose that each task realized in the system by a user is realized by means of an activity, we can prevent illegal information flows between users by preventing illegal information flows between activities. We must thus ensure that an activity with a high security level cannot transmit information to an activity with a lower security level (we will define in the next section how a security level is assigned to an activity). Yet, two activities may exchange information by accessing the same object (one activity writes an object and the other activity reads this object). Thus, we have to control in our system all interactions between activities and objects, i.e., to control each request accessing an object.

In order to realize these access controls, we have to assign security levels to the different entities of our model. We basically assign a security level to each user of the system. As the objects are the containers of information of the system, we have to assign them a security level (we will differently label stateless and stateful objects). In order to control each interaction between activities and objects, we have to assign security levels to the different requests of our system.

4.2 The different labels

We define a finite set of security levels partially ordered with respect to a binary relation "$\leq$".

Users:

A label is associated to each user of the system. This label represents the user's clearance.

Objects:

We label in a different way the stateless and the stateful objects. A label L_O is associated to each stateful object O. This label is the classification of the object. It represents the classification of the data that compose the state of the object. This classification is fixed and cannot be changed during the object lifetime.

Two labels $Llow_O$ and $Lhigh_O$ are associated to each stateless object O ($Llow_O \leq Lhigh_O$). $[Llow_O, Lhigh_O]$ is a *confidence interval* defining the trust one can put on the object. As a matter of fact, a stateless object does not store any application data, and thus cannot be assigned a classification as a stateful object. Nevertheless, each stateless object may be accessed by an activity like a stateful object. Each activity carries information with a particular security level. Each stateless object may thus potentially access this information. The labels we assign to a stateless object represent the trust one can put in this object, i.e., the trust that one can put on the actions that the object, as active entity, can execute. The high label represents the highest security level of data that can be read by the object. The low label represents the lowest security level of data that could be written by the object. Let us imagine, for example, that a stateless object is in fact a Trojan horse that keeps a local copy of all the data that are carried by all the requests accessing it. The low label of the object will guarantee that the copy will not be created with a label inferior to this label. This means that a future, malicious activity will need a clearance superior or equal to this low label to read the data. Conversely, the Trojan horse cannot read and then store data with a level highest than $Lhigh_O$. Let us imagine an administrator of a system who decides to use a new freeware NFS server that he gets from an anonymous ftp server. The administrator then estimates the trust that he can put in this NFS server (which is supposed to be a stateless object) and thus assigns it a confidence interval representing this trust.

Requests:

Just as we said in the previous paragraph, each activity in the system carries information and exchanges information with the objects it accesses. Thus, an activity must be labeled. We assign a *parenthesis* of labels (i.e., two labels) to each activity a. These labels are noted $Llow_a$ and $Lhigh_a$. These labels are floating labels and may change according to the type and the labels of the objects accessed by the activity. The low label represents the classification of the information carried by the activity, i.e., the highest level of the information that has been previously read by the activity. The high label represents the clearance of the activity. This clearance is initialized with the clearance of the user who started this activity. The parenthesis can then be noted: [*classification, clearance*]. The information carried by an activity is in fact carried by the different requests among the objects of the system it accesses. Thus, in order to make access controls in our model, we have to label these requests. We assign a *parenthesis* of fixed labels to each request r. These labels are noted $Llow_r$ and $Lhigh_r$. Let us imagine that an activity is composed of n requests $(r_1, r_2, ...r_n)$. The first request r_1 accesses an object in order to provoke the execution of one of its methods. This execution leads to the sending of a new message, the request r_2. The parenthesis of labels of r_2 will depend on the parenthesis of labels of r_1 and on the type and labels of the object accessed by r_1. Thus, the parenthesis of the different requests will be different according to the different objects accessed by the activity. The labels of the activity $[Llow_a, Lhigh_a]$ (which are floating labels) will successively be $[Llow_{r_1}, Lhigh_{r_1}]$ ... $[Llow_{r_n}, Lhigh_{r_n}]$.

4.3 The access rules

Our authorization scheme is composed of mandatory access rules based on the different labels we have just presented. These rules cannot be bypassed.

4.3.1 Notion of read access and write access

In the rest of the paper, the term of *read access* and *write access* must be understood in a very precise sense. We call read access on a stateful object O, any execution of a method of O which provokes an information flow from the internal state of O to the activity which executes the method. Reciprocally, we call write access any execution of a method of O which provokes an information flow from the activity which executes the method to the internal state of O. We will call read-write access any execution of a method of O which provokes an exchange of information between the activity which executes the method and the state of O (it is not worth considering "null access", i.e. methods which provoke no information flow between a stateful object and the request).

4.3.2 Access Rules

Access to stateless objects:

If a request r may access a stateless object O (through invoking any of its methods) then the intersection of $[Llow_O, Lhigh_O]$ and $[Llow_r, Lhigh_r]$ is not empty. (R1)

Access to stateful objects:

With respect to stateful objects, the security policy is enforcing the following rules (these rules derive from the Bell-LaPadula security policy rules):

simple security condition:
If a request r may read a stateful object O, then $L_O \leq Lhigh_r$. (R2)

***-property:**
If a request r may write a stateful object O, then $Llow_r \leq L_O$. (R3)

To enforce these rules, each method of a stateful object is assigned an attribute indicating which kind of access is realized by the method: read, write or read-write.

4.4 The access control scheme

In this section we consider the current request r of an activity a which accesses an object O.

- If O is a stateless object:
  - If $Lhigh_r < Llow_O$ or $Lhigh_O < Llow_r$:
    the access is denied. (R4)
  - If $Llow_O \leq Lhigh_r$ and $Llow_r \leq Lhigh_O$:
    the access is authorized with restriction: $[Llow_a, Lhigh_a]$ is changed to $[Max(Llow_a, Llow_O), Min(Lhigh_a, Lhigh_O)]$. (R5)

  This restriction means that the next request r' of the activity a will be labeled $[Llow_{r'}, Lhigh_{r'}] = [Max(Llow_r, Llow_O), Min(Lhigh_r, Lhigh_O)]$. Note the particular case: if $[Llow_O, Lhigh_O] \subset [Llow_a, Lhigh_a]$, then the access is authorized without restriction since $Max(Llow_a, Llow_O) = Llow_a$ and $Min(Lhigh_a, Lhigh_O) = Lhigh_a$.

  Rule (R4) guarantees that the request may not access objects unless the intersection of the request's parenthesis and the object's confidence interval is not empty. Rule (R5) describes the evolution of the parenthesis of an activity accessing an object O by a request r. If an activity accesses a stateless object whose low label dominates the current classification of the information carried by the activity then this current classification has to be set to this label. If an activity accesses an object whose high label is dominated by the clearance of the activity then the clearance of the activity has to be set to the high label of the object.

- If O is a stateful object:
  - If method m is a read method and if $Lhigh_r < L_O$:
    the access is denied. (R6)
  - If method m is a read method and if $L_O \leq Lhigh_r$:
    the access is authorized with restriction:
    $Llow_a$ is changed to $Max(Llow_a, L_O)$. (R7)
    Note the particular case: if $Max(Llow_a, L_O) = Llow_a$, the access is authorized without restriction.
  - If method m is a write method and if $L_O < Llow_r$:
    the access is denied. (R8)
  - If method m is a write method and if $Llow_r \leq L_O$:
    the access is authorized with no restriction. (R9)
  - If method m is a read-write method and if L_O is not in the interval $[Llow_r, Lhigh_r]$:
    the access is denied. (R10)
  - If method m is a read-write method and if L_O is in the interval $[Llow_r, Lhigh_r]$:
    the access is authorized with restriction:
    $Llow_a$ is changed to $Max(Llow_a, L_O)$. (R11)
    Note the particular case: if $Max(Llow_a, L_O) = Llow_a$, the access is authorized without restriction.

Rules (R6) and (R7) describe the evolution of the parenthesis of an activity making a read access to a stateful object. An activity must not read a stateful object whose classification dominates the high label of the activity (R6). If an activity makes a read access to an object whose classification dominates the classification of the information carried by the activity (i.e., the low label), then this low label has to be set to the classification of the object (R7). There is no restriction to be applied to the activity's parenthesis when an activity makes a write access to a stateful object. As a matter of fact, the sensitivity of the information carried by the activity does not change during such an access: there is only an information flow from the activity to the object. So, whether the access is authorized, or it is denied, no restriction is to be applied. An activity can always make a write access to a stateful object whose classification dominates the classification of the information carried by the activity (R9) and an activity cannot make a write access to a stateful object whose classification is dominated by the classification of the information carried by the activity (this corresponds to the *-property of the Bell-LaPadula model) (R8).

4.5 An example of an authorization scheme

In the following example, we consider that a SECRET user U wants to print an UNCLASSIFIED file f_3 on printer p_4 whose confidence interval is [UNCLASSIFIED, CONFIDENTIAL]. The set of objects which take place in the execution of this action are represented in Figure 3. U is a user (a person) of the system, ps_1 a print server of class *PrintServer*, fs_2 a file server of class *FileServer*, f_3 a file of class *File*, p_4 a printer of class *Printer* and tf a transient file located on the site of ps_1.

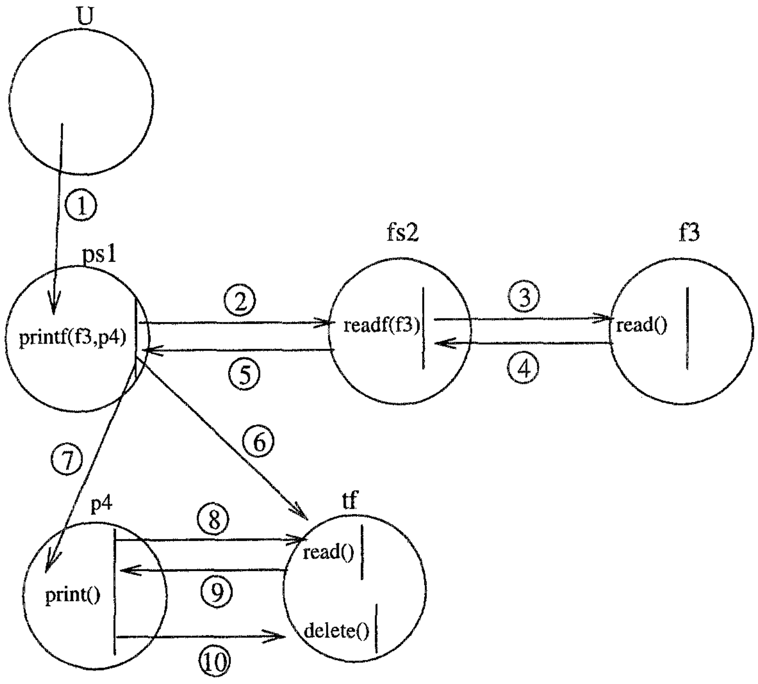

Fig. 3. Objects cooperation

The hierarchy of labels used in our example is represented below just as the labels of the different components of the system:

UNCLASSIFIED < CONFIDENTIAL < SECRET < TOPSECRET.

User U	: SECRET
$\mathbf{ps_1}$	: [CONFIDENTIAL, SECRET] (stateless object)
$\mathbf{fs_2}$	: [UNCLASSIFIED, SECRET] (stateless object)
$\mathbf{p_4}$	: [UNCLASSIFIED, CONFIDENTIAL] (stateless object)
$\mathbf{f_3}$	: UNCLASSIFIED (stateful object with methods *read*(), *write*(), *update*() and *delete*())

The scenario can be summarized as follows:

- User U logs in the system with a SECRET clearance. The activity a starts executing the first request for U. This request is the invocation of method *printf* of print server ps_1. This first request (**message 1**) is labeled [UNCLASSIFIED, SECRET]. The low label is UNCLASSIFIED because the activity does not hold any sensitive information. The high label represents the clearance of the user U.
- Print server ps_1 is a stateless object labeled [CONFIDENTIAL, SECRET]. The intersection between the parenthesis of labels of the request and the confidence interval of ps_1 is not empty, but as $Llow_r < Llow_{ps_1} \leq Lhigh_r \leq Lhigh_{ps_1}$, the access is authorized with restriction: $Llow_a$ is changed to CONFIDENTIAL, i.e., the parenthesis of the activity is changed to [CONFIDENTIAL, SECRET] (cf. R5).
- The activity goes on by invoking method *readf* of file server fs_2 (**message 2**). This request is labeled [CONFIDENTIAL, SECRET]. fs_2 is a stateless object labeled [UNCLASSIFIED, SECRET]. The access is thus authorized without restriction (cf. R5).
- The activity then makes an access to method *read* of f_3 (**message 3**). This request is labeled [CONFIDENTIAL, SECRET]. f_3 is a stateful object which is labeled UNCLASSIFIED and the access is a read access. The access is thus authorized without restriction (cf. R7).
- The activity then returns to fs_2 and then to ps_1 in two replies labeled [CONFIDENTIAL, SECRET] (**messages 4 and 5**). These two replies are authorized without restriction because their parenthesis of labels is respectively included in the confidence interval of fs_2 and ps_1 (particular case of R5).
- ps_1 creates a temporary file tf in which it copies f_3 (**messages 6**). This creation request is part of the same activity and thus must be labeled [CONFIDENTIAL, SECRET]. The file tf is a stateful object created with label $L_{tf} = Llow_r$ [1]. ps_1 makes a write access to tf with a [CONFIDENTIAL, SECRET] request. tf is a stateful object and $Llow_r \leq L_{tf}$, thus the access is authorized with no restriction (cf. R9).
- The activity then accesses printer p_4 through method *print* (**message 7**). The request is labeled [CONFIDENTIAL, SECRET], p_4 is a stateless object and the intersection between the parenthesis of labels of the request and the confidence interval of p_4 is not empty. But, as $Llow_{p_4} \leq Llow_r \leq Lhigh_{p_4} < Lhigh_r$, the access is authorized with restriction: $Lhigh_a$ is changed to CONFIDENTIAL (cf. R5).
- p_4 then invokes method *read* of tf (**message 8**). The request is now labeled [CONFIDENTIAL, CONFIDENTIAL] (because of the previous access to p_4), tf is a stateful object, the invoked method is a read method and $L_{tf} = Llow_r$, the access is thus authorized without restriction. The return of this request (**message 9**) is labeled [CONFIDENTIAL, CONFIDENTIAL] and is authorized because this parenthesis of labels is included in the confidence interval of ps_1. After printing, p_4 deletes file tf (**message 10**) in a [CONFIDENTIAL,

[1] The label of a transient stateful object is defined by a parameter of the creation request. Its level must be higher than or equal to the lower label of the request.

CONFIDENTIAL] request. The access is authorized with no restriction because $Llow_r = L_{tf}$ (delete is a write access).

5 Preventing the illegal information flows

5.1 Proof

This section is dedicated to the demonstration that our model prevents the illegal information flows. We describe here how we prevent these illegal information flows thanks to the set of rules that we have presented in the previous sections.

Preventing the illegal information flows consists in demonstrating that there does not exist a way for a user to infer information whose classification dominates the clearance of the user. This information inference might imply the collaboration of several users and the handling of several objects in the system. This can be expressed in the following way :

We want to verify that if s is a user, O a stateful object and $L_s < L_O$, then there do not exist series $\{i_1, i_2, ...i_l\}$ and $\{j_1, j_2, ...j_m\}$ such as:

$(s_{j_1}, O_{i_1}, read) \wedge (s_{j_1}, O_{i_2}, write) \wedge (s_{j_2}, O_{i_2}, read) \wedge \cdots \wedge (s_{j_m}, O_{i_l}, read)$.
with $s_{j_m} = s$ and $O_{i_1} = O$.

In the context of our model, this means that there do not exist series of activities and objects whose collaboration allow illegal information flows. Formally, this means, that there do not exist series $\{i_1, i_2, ...i_l\}$ and $\{k_1, k_2, ..., k_n\}$ such as:

$(a_{k_1}, O_{i_1}, read) \wedge (a_{k_1}, O_{i_2}, write) \wedge (a_{k_2}, O_{i_2}, read) \wedge (a_{k_2}, O_{i_3}, write) \wedge \cdots$
$\wedge (a_{k_n}, O_{i_l}, read)$ (a_{k_i} is an activity; the last one, a_{k_n} is started by user s_{j_m})

$(a_{k_1}, O_{i_1}, read)$ implies:
. $L_{O_{i_1}} \leq Lhigh_{a_{k_1}}$ (Rule R1)
. $Llow_{a_{k_1}}$ is changed to $Max(Llow_{a_{k_1}}, L_{O_{i_1}})$ (Rule R7)
$(a_{k_1}, O_{i_2}, write)$ implies:
. $Llow_{a_{k_1}} \leq L_{O_{i_2}}$ (Rule R9).
(with $Llow_{a_{k_1}}$ changed according to the access to O_{i_1}).

This leads to: $L_{O_{i_1}} \leq L_{O_{i_2}}$ and thus to: $L_{O_{i_1}} \leq L_{O_{i_2}} \cdots \leq L_{O_{i_l}}$

On the other hand, $(a_{k_n}, O_{i_l}, read)$ implies: $L_{O_{i_l}} \leq Lhigh_{a_{k_n}}$. Furthermore, as the activity a_{k_n} is started by the user s_{j_m}, we have the relation: $Lhigh_{a_{k_n}} \leq L_{s_{j_m}}$. This leads to: $L_{O_{i_l}} \leq L_{s_{j_m}}$.

We thus obtain: $L_{O_{i_1}} \leq L_{O_{i_2}} \cdots \leq L_{O_{i_l}} \leq L_{s_{j_m}}$ and thus $L_{O_{i_1}} \leq L_{s_{j_m}}$, i.e., $L_O \leq L_s$ which is contrary to the hypothesis ($L_s \leq L_O$).

5.2 Validation process and covert channels

We have presented in the previous sections some important notions of our model: the stateless objects and the attributes assigned to each method of a stateful object (read, write, read/write). The flexibility of our model and its interest with respect to the Bell-LaPadula security policy depend on the fact that we can or cannot validate such entities in the system. Given that, our model needs some validation processes in order to verify that an object which is supposed to be stateless is actually stateless (i.e., that it does not keep in local variables some information carried by the requests that access it). The confidence interval that we assign to a stateless object represents the trust we have for this object to be really a stateless object, thanks to the validation process. In the same way, we need to validate that each method of a stateful object which pretends to be a read (respectively write, read/write) method is actually a read (respectively write, read/write) method. We think that techniques like program proof, code analysis would quite easily allow to make such verifications. As a matter of fact, the properties that we want to check are quite simple. In order to check that a stateless object is actually a stateless object, we have to check that it does not keep in memory any information exchanged with the activities which access it. We can make this verification by checking that a stateless object reinitializes all its local variables after each method invocation. In order to check that a read method (for instance) is actually a read method, we have to check that the method only reads information from the activities which access it but does not give any information to these activities. If the messages are sent via sockets for example, we thus have to check all socket handlings in the source code of the method and check that all these handlings consist in reading information on the sockets.

In the same say, we must validate each object of the system to check that it cannot use a covert channel. A secure computer system ideally would not allow communication channels to exist. But eliminating all the covert channels in a secure system is in fact a quasi-impossible task: as soon as some resources of the system are shared, there are covert channels. Furthermore, eliminating all the covert channels in a system may lead to a non-responsive, non-reliable system. Given that, it is generally better to find ways to minimize illicit information leakage through such covert channels. That is the approach chosen in this paper. The model we present does not eliminate the existence of covert channels but aims at limiting the use of these covert channels: we consider that we can validate each object introduced in the system in a way that we can trust it not to try to transmit information using a covert channel. This validation is realized "off-line", i.e., before introducing the object in the system. This validation concerns all the objects of the system, stateless objects as well as stateful objects. One could pretend that this task seems quite difficult. We think it is not. As a matter of fact, it is commonly agreed that it is possible to identify all the covert channels that may exist on a system and that it is possible to measure their brandwidth. The B3 evaluation [2] of a system requires this identification and these measures, and B3 and even A1 systems do exist. So, if it is possible to identify the covert channels of a system, we assert that it is then possible to detect in a source code, the mechanisms that may try to exploit these covert channels (such analysis even seems easier that the analysis of the existence of covert channels).

If we consider that we cannot trust our validation process (i.e., methods to assign the confidence interval to stateless objects, mechanisms to check that a read (respectively write, read/write) method is really a read (respectively write, read/write) method and processes that verify that an new object introduced in the system does not try to transmit information through covert channels), a pessimistic solution is to consider that all objects are stateful (a single label is thus assigned to each object), that all the operations in the system are read/write and that all the activities have a parenthesis reduced to: $Llow_a = Lhigh_a$. This leads us to use the classical Bell-LaPadula model directly applied to distributed object systems.

6 Related work

One of the main characteristics of our model is that we always try to assign an activity the classification that actually represents the classification of the information carried by the activity. This allows to avoid the over-classification of the information because objects that are created are assigned a label that actually reflects the sensitivity of their state. The floating labels proposed by John Woodward (see Section 3.1) aim at implementing the same property even if this implementation does not address object systems.

This idea has also been exploited in object oriented databases. The SODA model [11] is based on the notions of object and method activation. In SODA, an object may have a label protecting the whole object or a set of labels protecting independently each attribute of the object and the whole object itself. Each method activation (the active entity in SODA) is assigned a clearance level and a current classification level. The clearance level is an upper bound for the current classification level and this current level can raise according to the objects accessed by the method activation. Just as in our model, the login method begins execution with classification level equal to SYSTEM LOW. The security policy in the SODA model is defined by rules that are similar to the rules we have presented here except one: a method may not modify an object unless the current classification level of the method is dominated by the level of the object and after the modification, the current classification level of the method becomes equal to the level of the object. We think that this modification of the current classification level of the method is not justified. As we said in the previous section, we think that it is possible to verify that some modifications are strictly write accesses (i.e., information flows from the method activation to the object) and that, in such cases, there is no reason to increase the current classification level of the method activation. Furthermore, the way of labeling the objects and particularly the attributes of the objects in SODA is only well suited to database systems and corresponds to a client/server model rather than a model based on cooperative objects. We think that this granularity of protection does not correspond to object oriented systems in which the protection rather addresses the whole state of the object. Thus, we think that it may be quite difficult to adapt SODA's model to distributed object systems.

In the message filter object-oriented model, Jajodia and Kogan [12] choose to assign one label to each object. Objects can exchange information only by sending messages. Thus, the principle of the model is to control all information flows by mediating the flow of messages. The message filter takes appropriate action upon intercepting a message and examining the classification of the sender and the receiver. Each method activation is assigned a level given by a variable *rlevel*. Jajodia and Kogan explain that the intuitive significance of *rlevel* is that it keeps track of the least upper bound of all objects encountered in a chain of method invocations, going back to the root of the chain (this *rlevel* corresponds to the classification of an activity of our model). This model has also some additional features in order to eliminate some timing channels. In [13], this model has also been proposed, in the context of a discretionary system. Even if this model is an interesting adaptation of the Bell-LaPadula model to object systems, we think that its main drawback is that it uniformly considers the objects of the system (and thus assigns a single label to stateless objects). This model is thus as restrictive as the Bell-LaPadula model in classical systems.

7 Conclusion and future work

The Bell-LaPadula security model prevents illegal information flows in a multilevel system but is too restrictive. We have proposed in that paper a multilevel security model that is derived from the Bell-LaPadula model but that is less restrictive (thanks to the notion of stateless objects) and that is adapted to the distributed object systems.

It should be interesting to study the adaptation of other security policies to distributed object systems. For example, we think it an interesting future work to study the adaptation of Biba's [14] integrity policy to object systems.

References

1. D. Bell and L. LaPadula, "Secure Computer Systems: unified Exposition and Multics Interpretation," Tech. Rep. MTR-2997, MITRE Co., July 1975.
2. "U.S. Departement of Defense Trusted Computer Security Evaluation Criteria (TCSEC)." 5200.28-STD, December 1985.
3. C. Landwehr, "Formal Models for Computer Security," *ACM Computing Surveys*, vol. 3, pp. 247–278, September 1981.
4. E. R. Lindgreen and I. Herschberg, "On the Validity of the Bell-LaPadula Model," *Computers and Security*, vol. 13, pp. 317–333, 1994.
5. J. McLean, "Reasoning about Security Models," in *Proc. of Symposium on Research in Security and Privacy, IEEE Computer Society Press*, (Oakland, California(USA)), pp. 123–131, 1987.
6. D. Bell, "Concerning 'Modeling' of Computer Security," in *Proc. of Symposium on Research in Security and Privacy, IEEE Computer Society Press*, (Oakland, California(USA)), pp. 8–13, 1988.
7. J. Woodward, "Exploiting the Dual Nature of Sensitivity Labels," in *Proc. of Symposium on Research in Security and Privacy, IEEE Computer Society Press*, (Oakland, California(USA)), pp. 23–30, 1987.

8. L. Fraim, "Scomp: A Solution to the Multilevel Security Problem," *IEEE Computer*, vol. 16, pp. 26–34, July 1983.
9. B. d'Ausbourg, "Implementing Secure Dependencies Over a Network by Designing a Distributed Security Subsystem," in *Proc. of European Symposium on Research in Computer Security*, (Brighton(UK)), pp. 249–266, November 1994.
10. J. Banino, J. Fabre, M. Guillemont, G. Morisset, and M. Rozier, "Some Fault-Tolerant Aspects of the Chorus Distributed System," in *Proc. of 5th International Conference on Distributed Computing Systems*, (Denver, Colorado), pp. 430–437, May 1985.
11. T. Keefe, W. Tsai, and M. Thuraisingham, "SODA: a Secure Object-oriented Database System," *Computers and Security*, vol. 8, no. 6, pp. 517–533, 1989.
12. S. Jajodia and B. Kogan, "Integrating an Object-Oriented Data Model with Multi-Level Security," in *Proc. of the 1990 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 48–69, May 1990.
13. E. Bertino, P. Samarati, and S. Jajodia, "High Assurance Discretionary Access Control for Object Bases," in *Proc. of 1st ACM Conference on Computer and Communications Security*, (Fairfax, Virginia (USA)), pp. 140–150, November 1993.
14. K. Biba, "Integrity Considerations for Secure Computer Systems," Tech. Rep. ESD-TR 76-372, MITRE Co., April 1977.