Using Security Methods to Enforce Mandatory and Discretionary Access Control in an Object Database

James M. Slack
Department of Computer and Information Sciences
Mankato State University
Mankato, Minnesota 56001, USA
slack@mankato.msus.edu

Abstract

In this paper, we propose a new security enforcement mechanism and demonstrate how this mechanism can enforce policies for both mandatory access control (MAC) and discretionary access control (DAC) in an object database system. Each class may have a security method that can block messages that leave instances of the class, and can block messages directed to instances of the class. Each superclass controls the invocation of security methods in its subclasses. A subclass cannot override the security methods in its supersclasses, but the subclass can extend those security methods. The mandatory access control policy is enforced by a single security method along with other methods in the highest class in the class hierarchy. The discretionary access control policy is enforced by a single security method along with other methods a subclass of the highest class.

Keywords Object-oriented databases, security and integrity, security methods, discretionary access control, mandatory access control

1 Introduction

In [15], we proposed a security enforcement mechanism based on a simple extension to the object database model: the addition of security methods. In [16], we demonstrated how to use security methods to enforce discretionary access control (DAC) in an object database. Many database systems use a discretionary access control policy to protect information. Discretionary access control is based on ownership of objects [14, 2]. The owner of an object can grant permission to retrieve, change, and delete the object. In this paper, we show how to incorporate mandatory access control (MAC) in an object database.

A security method enforces this general, hierarchical security policy: a higher level class controls messages to or from its descendants [15].

Whenever an object sends or receives a message, the system invokes the security method in the topmost class, which can then allow or disallow the message. Ultimately, the security method in the topmost class has control over invocation of all other security methods. Automatic invocation of this security method ensures that it is invoked for all messages, and thus cannot be circumvented. Later in this paper, we will use this feature to design a mandatory access control mechanism.

To assist in this determination, the topmost security method may invoke a security method in a subclass. This subclass must be an ancestor of the source or destination of the message. A security method may, in turn, invoke a security method in a subclass, thus starting a chain of security method invocations. Therefore, the topmost class decides to invoke the security method of the message sender's ancestor, the message receiver's ancestor, or both. These ancestors are then responsible for deciding which of their descendant's security methods to invoke.

For example, in Figure 1, objectX is sending a message to objectY. The system will invoke the security method in TopClass, the highest class in the hierarchy. TopClass's security method may decide whether to allow the message, or it may delegate the decision to a subclass. To delegate the decision, TopClass's security method would invoke the security method in ClassZ. However, TopClass cannot invoke the security method in ClassV, as ClassV is not an ancestor of either objectX or objectY. ClassZ's security method may invoke the security methods in ClassX and ClassY.

Figure 1: A class hierarchy

Figure 2: MAC and DAC enforcement

The desired security policies determine the criteria for selecting which subclass security methods to invoke. The database product vendor or local system security officer can define rules of message control for particular circumstances. For example, to enforce traditional mandatory access control (MAC), the system security officer could own the top classes in the hierarchy, as shown in Figure 2. The security method in the second level class enforces MAC, while the security method in the third highest class enforces DAC. The MAC security method would invoke the DAC security method only when the message does not violate MAC policy.

Security methods are a generalization of Jajodia and Kogan's message filter [9]. (Thomas and Sandhu extend the message filtering approach in [18].) Their message filter approach does not allow a message to flow from object to object directly. Instead, the message must pass through a message filter. The message filter lets the message go through only if the information would not flow from a high level to a low level object or subject. In our model, the set of security methods invoked for the message is analogous to Jajodia and Kogan's message filter. Security methods allow extension of the message filtering mechanism: A user can create a new class that implements some new security policies. The system will enforce these new policies if security method in the superclass invokes the security method in this new class.

Our approach is similar to work done by Fernandez, who proposed an access control mechanism that follows the class hierarchy [7]. In his approach, an authorization rule is a triple $(U, A, AO)$ where $U$ is a user (or group of users), $A$ is an access type (such as read or write), and $AO$ is an attribute (or set of attributes) of an object. The authorization mechanism in our proposal also follows the class hierarchy, but there is more control over the inherited rules with the Security requirement predicate. In our proposal, there is no outside access to attributes, only to methods. Therefore, our approach provides additional data independence.

2 Object Database Model

We assume an object database model based on Cattell's text [3] and Kim's essential features [10]. The database captures information about a real-world entity as an object. The system automatically assigns each object a unique identifier when it creates the object. Each object encapsulates a set of hidden attributes that define the object's state and a set of public methods that define the object's behaviors. Each attribute value may be a value from a primitive class (such as real, integer, or string), an object identifier, or a collection of object identifiers.

A class is a template for a group of similar objects, and individual objects are instances of their respective classes. (In our model, a class is also an object.) Each object belonging to a particular class has its own set of attribute values but shares the same set of methods with other objects in its class. For this paper, we assume single inheritance—a subclass can have no more than one parent class. Therefore, classes are organized in a generalization/specialization hierarchy through inheritance. An ancestor class is a parent class or superclass; a descendent class is a subclass. A subclass may redefine an inherited method—this is called overriding the method.
We assume that there is a single topmost class TopClass that is most general in the specialization hierarchy. TopClass defines the methods newClass(), newInstance(), newInstanceMethod(), newClassMethod(), and newSecurityMethod() for creating new classes, instances, and methods. Each class inherits these methods from TopClass.

There are two kinds of methods: instance methods and class methods. An instance method is a method that applies to an instance of a class. For example, suppose the class Employee has a method called changeSalary(). This method should be an instance method so it can change the salary in an individual instance of Employee. A class method is one that applies to a class object. The methods of TopClass are examples of class methods.

Each method has an access mode associated with it. An access mode is the set of rights necessary for the method to execute. A user may invoke a method if the user's rights are a superset of the method's access mode. Likewise, a method x may invoke another method y if x's rights are a superset of y's. These access mode restrictions ensure, for example, that a method that has permission to write to an object does not invoke a message that can read from an object. In this paper, we will assume that an access mode can be r (read), w (write), or rw (read/write). We assume that the database system will make sure that a method with read access mode does not change the attributes in any object, and that a method with write access mode does not read any attribute.

We assume that the system stores methods (including security methods) in the database, not in applications. Thus, the security enforcement mechanism protects both methods and objects. We assume that the operating system guarantees safe transmission of messages, that is, that users cannot circumvent or spoof normal message transmission. Our model assumes run-time binding of messages to methods, as in Smalltalk [8].

We assume that the operating system provides appropriate protection for the database management system, and that the database system already does identification and authentication.

3 Security Methods

Each class may have a security method. Let us use TopClass.security() as the name of the security method in TopClass. The database system invokes TopClass.security() for a message involving any instance or class in the class hierarchy. TopClass.security() determines whether to allow or disallow this message.

TopClass.security() may be defined to invoke two other security methods to assist in authorizing the message. One of these is source.security(), the next security method on a path from TopClass to the class of the message source. If there is no other security method on the path, then source.security() returns true. The other security method, dest.security(), works in the same manner with the destination of the message.

For example, in Figure 3, TopClass may invoke ClassV.security(), because ClassV is on path from TopClass to the message source. Likewise, TopClass can invoke ClassZ.security(), because ClassZ is on a path from TopClass to the message destination.

TopClass can invoke only one other security method if the message source and destination share a common descendant of TopClass. For example, in Figure 1, the two paths from TopClass share ClassZ. Therefore, the only security method that TopClass.security() may invoke is ClassZ.security().

Each class C is the root of a subset of the class hierarchy—let us call this subset C's subhierarchy—and each class C may have a security method. The superclass of C may invoke C.security() for a message involving any instance or class in C's subhierarchy. Thus, C.security() controls messages in C's subhierarchy much as TopClass.security() controls messages in the whole class hierarchy. The difference between TopClass.security() and C.security() is that the system in-
vokes `TopClass.security()` for every message, whereas C’s superclass determines whether or not to invoke `C.security()`.

Each security method can retrieve (but not change) the following information about a message.

- **source**: The object sending the message
- **dest**: The object receiving the message
- **user**: The current user
- **method**: The method the message is invoking
- **args**: The list of actual parameters for the method, accessed using `arg(name)`, where `name` is the name of the formal parameter

The security method may use this information to determine whether to allow the message.

Each security method consists of a set of one or more security method rules, which are in no particular order. We write a security method rule in a tabular, declarative style, as shown in Figure 4. Each security method rule consists of the following parts.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Rule type</th>
<th>User (or group)</th>
<th>Message source</th>
<th>Message destination</th>
<th>Security requirement</th>
<th>Methods/rights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>any</td>
<td>any</td>
<td></td>
<td><code>true</code></td>
<td>any</td>
</tr>
</tbody>
</table>

**Figure 4**: An empty security method rule

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Rule type</th>
<th>User (or group)</th>
<th>Message source</th>
<th>Message destination</th>
<th>Security requirement</th>
<th>Methods/rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Permit</td>
<td>any</td>
<td>any</td>
<td></td>
<td><code>source.security() AND dest.security()</code></td>
<td>any</td>
</tr>
</tbody>
</table>

**Figure 5**: The default security method

### 4 Mandatory Access Control

Mandatory access control is concerned with making sure that only authorized users are able to access sensitive information [6]. Each object `o` is assigned a security level `I(o)`. The security level of a subject `s`, denoted `I(s)`, is called the clearance of that subject. A subject’s clearance affirms the authorization of that subject to access objects at a certain security level.

Each security level is composed of two parts: a sensitivity level and a compartment set. The sensitivity level signifies the effect of disclosure on national security, for example Unclassified, Classified, Secret, and Top Secret. The compartment set consists of topic areas such as Nuclear and Spy.

The combination of sensitivity level and compartment set is called the classification or security level of an object. Sensitivity levels are totally ordered and compartment sets can be partially ordered with the subset relation. Therefore, classification is also partially ordered. Suppose two objects have classifications `(a, b)` and `(c, d)`, where `a, c` are sensitivity levels and `b, d` are compartment sets. Then `(a, b) ≤ (c, d)` if `a ≤ c` and `b ⊆ d`. The relation ≤ on classifications forms a lattice if we assume a lowest classification of (Unclassified, no compartments) and a highest of (Top Secret, all compartments) [11, 5]. As an example, Figure 6 shows the lattice formed from the single compartment...
Spy and the two sensitivity levels Unclassified and Classified.

The Bell-LaPadula MAC model has been extended and restated several times since its first definition in the mid-1970s. Thomas and Sandhu summarize the Bell-LaPadula model as follows [18].

**Simple security condition** A subject $s$ can read from an object $o$ only if $I(s) \geq I(o)$.

***-property** A subject $s$ can write to an object $o$ only if $I(s) \leq I(o)$.

The common interpretation of the Bell LaPadula model consists of two rules: "no read up" (from the simple security condition) and "no write down" (from the *-property) [11]. For example, in Figure 7, user A is given clearance Secret (assume that all subjects and objects are in the same compartment for this example). User A may read and write to objects in the same classification (e.g., File2), may write to objects in classification Top Secret (e.g., File1), and may read from objects in classification Confidential (e.g., File3).

Along with the usual security levels, we will assume a security level `public` for objects that any user can access, such as primitive types (`integer`, `string`, etc.) and classes (`TopClass`). Only the system security officer (SSO) can create or write a `public` object, but any user can read it.

We now illustrate how to enforce the simple security condition and the *-property using security methods. We define `TopClass` to enforce mandatory access control with methods for creating classes, instances, and methods. We also include an attribute for keeping track of the security level (sensitivity level and compartment) of each object, along with a method for making sure each access is permitted. Here is the definition of `TopClass` in a C++-like syntax.
newClassMethod Creates a new class method in the class.

newSecurityMethod Creates a new security method in the class.

upgradeLevel Changes the security level of the object to a higher security level.

getLevel Returns the security level of the object.

The level attribute records the security level of the instance or class. TopClass.level is initialized to public so anyone can create a subclass of TopClass. (TopClass itself is an object, and so has a level attribute.)

Rule 1 in TopClass.security() allows the system security officer (SSA) to access any object with any method. This rule is especially necessary when the database is installed, because it allows the SSA to establish users and groups.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule type</td>
<td>Permit</td>
</tr>
<tr>
<td>User (or group)</td>
<td>SSA</td>
</tr>
<tr>
<td>Message source</td>
<td>any</td>
</tr>
<tr>
<td>Message destination</td>
<td>any</td>
</tr>
<tr>
<td>Security</td>
<td>true</td>
</tr>
<tr>
<td>Methods/rights</td>
<td>any/rw</td>
</tr>
</tbody>
</table>

Rule 2 enforces the simple security condition in the Bell-LaPadula model. A subject can invoke a read method in another object if the security level of the subject (source) is at least that of the object (dest). This rule also checks source.security() and dest.security(), allowing enforcement of DAC in a subclass of TopClass. (The function rights(method) returns the access rights of a method.)

<table>
<thead>
<tr>
<th>Rule number</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule type</td>
<td>Permit</td>
</tr>
<tr>
<td>User (or group)</td>
<td>any</td>
</tr>
<tr>
<td>Message source</td>
<td>any</td>
</tr>
<tr>
<td>Message destination</td>
<td>any</td>
</tr>
<tr>
<td>Security</td>
<td>rights(method) == r AND dest.getLevel() &lt;= source.getLevel() AND source.security() AND dest.security()</td>
</tr>
<tr>
<td>Methods/rights</td>
<td>any</td>
</tr>
</tbody>
</table>

Rule 3 enforces the *-property in the Bell-LaPadula model. A subject (user) can modify an object O1 (dest) in a manner dependent on an object O2 (source) only if the security level of O1 is at least that of O2.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule type</td>
<td>Permit</td>
</tr>
<tr>
<td>User (or group)</td>
<td>any</td>
</tr>
<tr>
<td>Message source</td>
<td>any</td>
</tr>
<tr>
<td>Message destination</td>
<td>any</td>
</tr>
<tr>
<td>Security</td>
<td>rights(method) == w AND source.getLevel() &lt;= dest.getLevel() AND source.security() AND dest.security()</td>
</tr>
<tr>
<td>Methods/rights</td>
<td>any</td>
</tr>
</tbody>
</table>

Rule 4 allows a subject to invoke any method—read or write—at that subject's clearance, subject to enforcement of DAC.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule type</td>
<td>Permit</td>
</tr>
<tr>
<td>User (or group)</td>
<td>any</td>
</tr>
<tr>
<td>Message source</td>
<td>any</td>
</tr>
<tr>
<td>Message destination</td>
<td>any</td>
</tr>
<tr>
<td>Security</td>
<td>rights(method) == rw AND source.getLevel() &lt;= dest.getLevel() AND source.security() AND dest.security()</td>
</tr>
<tr>
<td>Methods/rights</td>
<td>any</td>
</tr>
</tbody>
</table>

We define the class Subject as a subclass of TopClass. Each instance of Subject may represent a user, and allows that user to invoke methods in other objects. (A given user may connect to the system as different subjects at different levels, and may be represented by more than one subject.) A Subject instance encapsulates the user and that user's security clearance.

class Subject : private TopClass {
  public:
    void setClearance(Clearance newClearance) w;
    Clearance getClearance(void) r;
    void setUser(User newUser) w;
    User getUser(void) r;
    Variant invokeMethod((bf TopClass) object, Method method, List<Variant> arguments) rw;

  private:
    Clearance clearance;
    User user;
};

The methods in Subject include the following.

setClearance Sets the clearance of a subject.

getClearance Returns the clearance of a subject.
setUser Makes the subject represent a particular user.

getUser Returns the user that a subject represents.

invokeMethod Invokes a method on behalf of subject's user. The return type Variant can be any type.

The security method in Subject includes two rules: Rule 5 and Rule 6. Rule 5 allows only the user assigned to a Subject instance to invoke methods from that subject.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Rule type</th>
<th>User (or group)</th>
<th>Message source</th>
<th>Security requirement</th>
<th>Methods/rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Permit</td>
<td>any</td>
<td>any</td>
<td>user == this.user</td>
<td>invokeMethod/rw</td>
</tr>
</tbody>
</table>

Rule 6 allows another subject or object to find out about this subject. The MAC enforcement in TopClass.security() will prevent unauthorized access to this information.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Rule type</th>
<th>User (or group)</th>
<th>Message source</th>
<th>Security requirement</th>
<th>Methods/rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Permit</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>getClearance/r, getUser/r</td>
</tr>
</tbody>
</table>

5 Discretionary Access Control

We showed how to implement discretionary access control with security methods in [16]. This section briefly reviews our approach and shows how to integrate DAC with MAC enforcement. We define DacClass with methods for creating classes, instances, and methods. We also include methods for granting and revoking permissions to users and groups, and checking those permissions. Finally, we include access attributes—the owner of the object and the set of user/method permissions.

Here is the definition of DacClass in a C++-like syntax.

```cpp
class DacClass : public TopClass {
public:
    Boolean grant(User user,
                  String method) rw;
    Boolean revoke(User user,
                   String method) rw;
    Boolean revoke(UserGroup group,
                   String method) rw;
    Boolean okToUse(UserGroup group,
                     String method) r;
    void changeOwner(User user) r;
    User getUser(void) r;
private:
    User owner;
    Set<User, Method> permissions;
};
```

The methods in DacClass include the following.

grant Gives a user or group permission to use a particular method in the class or instance. If the method is any, then the user or group can use any method in the object.

revoke Revokes permissions from a user or group.

okToUse Checks whether a user has permission to use a method in the object.

changeOwner Changes the owner of the object to some other user.

getOwner Returns the owner of the object.

The owner attribute records the owner of the instance or class. The permissions attribute stores the user-method permission pairs. DacClass.owner is initialized to the system security officer.

We defined Rule 7 in DacClass.security() to allow the system security officer (SSA) to access any object with any method. However, Rule 1 makes Rule 7 redundant.

<table>
<thead>
<tr>
<th>Rule number</th>
<th>Rule type</th>
<th>User (or group)</th>
<th>Message source</th>
<th>Security requirement</th>
<th>Methods/rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Permit</td>
<td>SSA</td>
<td>any</td>
<td>true</td>
<td>any/rw</td>
</tr>
</tbody>
</table>

Rule 8 lets an object's owner grant and revoke permissions to access the object, and find out what permissions exist on an object. Rule 8 also lets an object's owner transfer the ownership rights to another user. After the previous owner has transferred these rights, that user may still have other explicit rights to the object. However, the new owner can revoke any of these remaining rights.
Rule number 8
Rule type Permit
User (or group) any
Message source any
Message destination any
Security dest.getOwner() == user
requirement
Methods/rights grant/rw, revoke/rw, ok-ToUse/r, changeOwner/w

Rule 9 allows the owner of a class to create instances and subclasses of that class. The function isClass() is a database-system primitive, and returns true if the argument is a class.

Rule number 9
Rule type Permit
User (or group) any
Message source any
Message destination any
Security isClass(dest) AND
requirement dest.getOwner() == user
Methods/rights newInstanceMethod/rw,
newClass/ rw

Rule 10 lets the owner of a class create instance methods for the class. The rule also ensures that the owner does not try to create a new method that overrides a method in DacClass, such as newClass(). If a user could override such a method, then the security enforcement mechanism may not protect descendants of that class. There is also a similar rule for newClassMethod().

Rule number 10
Rule type Permit
User (or group) any
Message source any
Message destination any
Security isClass(dest) AND
requirement dest.getOwner() == user
NOT IN {“newClass”,

Methods/rights newInstanceMethod/rw,
newClass/ rw

Rule 11 lets an object send a message to another object if the user has permission to do so. The user must have permission to invoke the particular method in the destination object from a particular source object.

Rule number 11
Rule type Permit
User (or group) any
Message source any
Message destination any
Security dest.okToUse(user,
requirement method)
Methods/rights any

6 Conclusion
We have shown how to enforce a simple mandatory access control policy using security methods. Our implementation may be particularly useful in a distributed environment, because each object stores its own access control information. Thus, this information travels with the object.

Our implementation of mandatory access control could be modified to use concurrency for write-up operations, as presented by Thomas and Sandhu [18]. In this approach, a write-up operation spawns a new process to reduce the possibility of a timing channel. We would like to explore an implementation of Clark-Wilson style integrity [4] for commercial applications, including temporal constraints such as separation of duty rules. Promising directions include Maiocchi’s time calculus [12] and McLean’s n-person rules [13].

Our implementation of a discretionary access control policy could be extended to include some of the features of the Orion/Itasca system, such as weak and strong authorizations, and negative authorizations [2].

Eventually, we would like to show that our security method approach can enforce any policy that can be expressed by Abrams’ Generalized Framework for Access Control [1]. In this framework, discretionary access control and mandatory access control are points on a large space of possible control policies. The framework is based on properties of subjects and objects, context, authorities, and formal expressions of security policy.

Slack and Unger defined a declarative integrity constraint language that could be incorporated in the proposed security mechanism [17]. This integrity constraint language is based on an extension to the object database model that is similar to security methods. The ability to express integrity constraints would further generalize security methods.

References


