Building a secure database using self-protecting objects

M S Olivier and S H von Solms
Department of Computer Science
Rand Afrikaans University
Johannesburg [*]

May 23, 1991

Abstract

In current database systems the responsibility for enforcing security is often given to the various application programs. Even where the Database Management System (DBMS) does supply security mechanisms, a single application program often handles sensitive transactions for some users and therefore needs a high clearance for accessing data—this may render the provided mechanisms inadequate. Furthermore, the user's identity is often concealed because the user has many 'software agents' acting on its behalf—especially in distributed environments. A simple mapping between subjects and objects is no longer possible.

We propose a model for extending Object-Oriented Database Systems to enable objects themselves to ensure security—ie to protect themselves. This extension is based on the concept of 'baggage'—baggage is collected from all components involved in any request; this baggage may then be verified by the object against its personal security profile before any method is executed.

Keywords: Security, Multilevel Secure Database, DBMS, Object-oriented, Path Context Model (PCM)

1 Introduction

This paper describes a model to build a secure object-oriented database based on the Path Context Model (PCM) [2,3,4]. We will refer to this model as SECDB (SECure DataBase).

Both the appeal and concerns regarding security of object-oriented databases are discussed in [11]. On the positive side object-orientation supports encapsulation, the possibility to model security in real-world terms and the potential of inheritance. On the negative side the main problem occurs when a class and its superclass have different sensitivity levels: this may allow information to flow from a higher sensitivity level to a lower level.

The necessary extensions to an object-oriented database to support security are considered. Such extensions must fit as cleanly as possible into the object-oriented model. The proposed mechanisms are designed to be objects themselves, conforming to the normal object, class and inheritance structures. Also, in line with the object-oriented database model, the security information must be stored as part of the database, together with the data and code.

We will mainly be concerned with the secrecy aspect of security, although the proposed model has some advantages regarding integrity.

The reader is referred to [8] for a discussion of the classical secure database implementation strategies.

[*]Address correspondence to Martin Olivier, Department of Computer Science, Rand Afrikaans University, PO Box 524, Johannesburg, 2000 South Africa; Email: Martin.Olivier@f1.n7101.z5.fidonet.org

2 Object-oriented extensions

This section introduces the terminology used. The concepts are treated in more detail in subsequent sections.

2.1 Object-oriented basics

We consider objects, classes, inheritance, methods and messages as the necessary ingredients of object-orientation [14, pp 8–11]. Also, in a secure system encapsulation [14, p 11] must be enforced—that is an object's instance variables must be hidden from all methods except the object's own methods.

2.2 Messages and baggage

Throughout this paper we use the term 'message' to refer to an active message: it starts to exist when it is sent (by a user or from some method which is being executed). We will say a message 'activates' a method; messages sent by the executing method are 'spawned' by the original message; the message directly responsible for sending a message will be referred to as the 'spawning' message.

Example

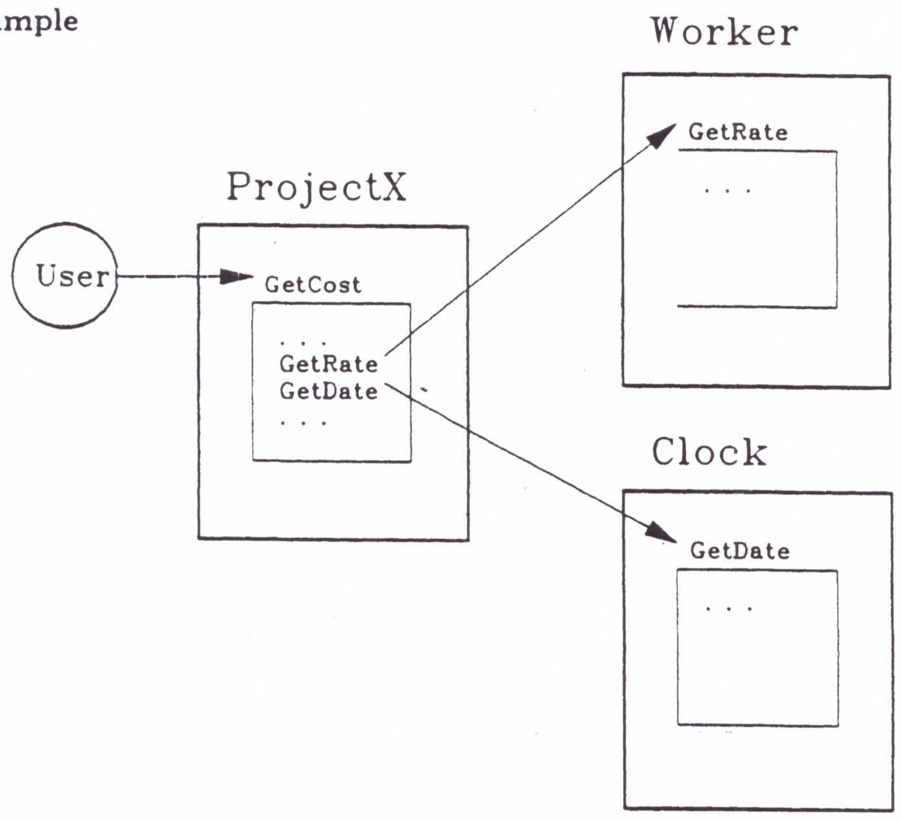

Figure 1: Messages illustrated

See figure 1. Suppose a user sends the message *GetCost* to an object *ProjectX*. The method *GetCost* sends the message *GetRate* to the object Worker and then *GetDate* to the object *Clock*. We say that *GetCost* spawned *GetRate* and *GetDate*. The spawning message of both *GetRate* and *GetDate* is *GetCost*. Note that the message *GetCost* starts to exist when it is sent by the user; *GetRate* exists until it returns an answer. After that *GetDate* starts to exist and ends its existance when it returns the date. After this the *GetCost* message stops to exist when it returns its answer. □

Messages are required to carry 'baggage' with them. SECDB is designed to support the baggage concept as defined in the Path Context Model (PCM) [2,3,4]. Informally, PCM requires that 'baggage' or information must be collected about all relevant software and hardware components involved in the handling of any request. A profile is associated with every item in the system to which access must be controlled—this profile specifies which components are required to take part in the handling of the request and which components are not allowed to take part. A profile may, for example, be defined to

1. Allow access only if one of a few specified users has originated the request; but
2. Deny access if some non-encrypted communications line has been used to transmit the request.

We expand on this later; initially this baggage may be viewed as the 'clearance level' of the sender or originator of the message: it is an indication of the trust associated with the request.

2.3 Profile objects

SECDB defines a profile as an object containing security information about protected information; given the baggage carried by a message, a profile can use its security information to determine whether the message may be allowed to access the protected information. The security information will typically be allowing and prohibiting contexts as defined in PCM; initially this information may be viewed as the 'sensitivity level' or 'classification level' of the information protected by the specific profile—ie a restriction of who is allowed to access the information. Profiles are normal objects—instances of profile classes—placing no additional requirements on the underlying system.

Since messages carry information they need protection and will therefore also have profiles associated with them. The information carried by a message will typically consist of parameters; however, the information can be much more subtle: the fact that a message has been sent can compromise information. Note that messages are active entities which cannot be accessed (assuming that the operating system protects executing programs). Messages therefore do not pose a security threat—the problem arises when such a message 'drops' information somewhere where it can be accessed, eg by assigning it to a variable or by creating an object. So, whenever a message is sent, profiles are taken along for all the information contained in it—contained directly in its parameters or contained implicitly. These profiles carried by messages will then be associated with any objects created or modified by that message. The profiles associated with a message will also be tagged to messages spawned by it. This will ensure that the *-property [1] is not violated when messages are sent between objects; simply put, this property requires that no user (or application) can write information he has access to where someone without the proper authorization can then read it. We will formalize this mechanism later. See section 5 for an example.

It is important to distinguish between baggage carried by messages and profiles carried by messages. Baggage is a record of the path a request has followed, ie which user initiated the request, which application programs were involved, which networks have taken part, etc. When a message arrives at an object, the object's profile(s) will look at this baggage to determine whether access should be granted to the message. On the other hand, a message also carries profiles with it to protect the information contained by the message. These profiles will have no influence on what the message may or may not access; the profiles will rather be attached to any variables modified or objects created by the message, controlling access to those variables and objects.

2.4 Gates

In SECDB a gate is a 'boundary' around every object in the system: whenever a message is sent to an object, the message will be intercepted by the sending object's gate and then by the receiving object's gate. The sending object's gate will tag any profiles associated with the sending object to the message. The receiving object's gate will ask access permission from the receiving object's profile(s). The profile will make this decision based on the baggage sent with the message and its internal profile information. If access is denied, it is the gate's responsibility to handle the rejection of the message. If access is allowed, the gate will associate any profiles carried by the message with the instance variables changed or created in the receiving object (in addition to any existing profiles).

3 Profiles

3.1 Description of profiles

Profiles will be defined for information in a SECDB database to which access is restricted. Profiles are objects containing access requirements: these requirements will typically consist of a list of objects (users, programs, etc) that may access the information and a list of objects that may not take part in a request to access the information. Based on the baggage carried by a message and these requirements in a profile the profile can determine whether the message satisfies the access requirements or not. PCM specifies that the access requirements must be in the form of a Random Context Grammar [13]: it will specify which contexts (ie human, software and hardware components) are required to be in the baggage and which contexts are not allowed to appear in the baggage.

The profile may also support methods to dynamically modify the profile data; however, if such mechanisms are too general, the resulting security may be too complex, reducing trust. We do not consider meaningful restrictions to methods supported by profiles in the current study.

SECDB additionally requires that a profile object provides a method *GrantAccess*, which may be called by the protected object's gate and which will reply with a *Yes* or a *No*.

Note that the PCM requirement of a Random Context Grammar profile may be modified to yield other interesting versions of SECDB; we discuss one such possibility later.

3.2 Profiles as objects

The implications of profiles being objects are

1. There are profile classes, subclasses and superclasses; and
2. Profile objects are protected by profiles themselves.

Considering (1), it is convenient to assume that a profile class has more restrictive access requirements than its (profile) superclass, ie that each subclass has a higher (or equal) 'sensitivity level' than its superclass. A hierarchy of profile classes then forms a partially ordered set of 'sensitivity levels'. In order to guarantee this, SECDB requires that *GrantAccess* in any profile subclass will send a message to *GrantAccess* in the superclass of the profile subclass; access will be denied if it is denied by the superclass.

From (2) it follows that there is potentially an infinite chain of profiles. This problem is easily (and soundly) solved by allowing one profile object to act as its own security profile—the problem is similar to the one caused by the fact that classes are indeed objects and therefore instances of some metaclass, which is again an object itself.

3.3 Associating profiles with information

In order to protect information, profiles are specified when a class is defined. The specified profile(s) will then be used for all instantiations of that class.

An object may be viewed as a layered entity: The outermost layer contains those instance variables and methods (and shares those class variables) defined in its class; the next layer those defined in the superclass of its class, and so on up to the innermost level which contains the items defined highest up in the class hierarchy. (In Smalltalk terminology [5], it contains the class variables and methods defined in class *Object*.) Figure 2 illustrates this. Since information at different levels in the class hierarchy may have different security requirements, different layers of an object may be protected by different profiles.

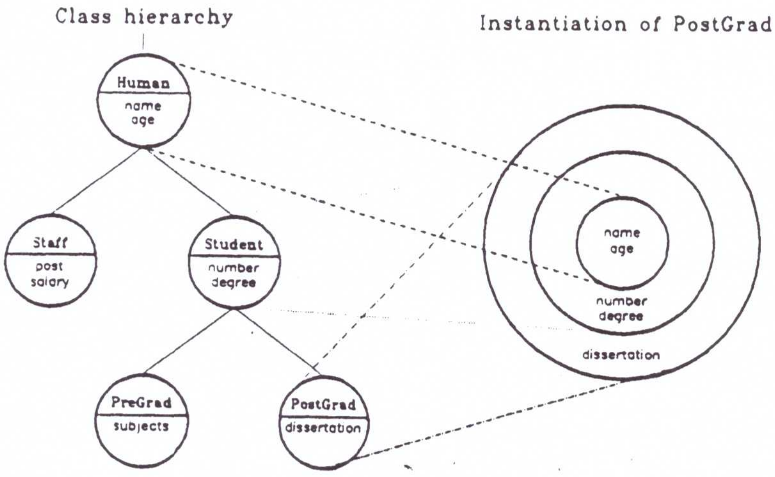

Figure 2: Objects are layered entities

A profile may be associated with one of more of the following:

1. A layer of an object;
2. Specific methods of an object; and/or
3. Specific instance variables of an object.

In order to activate a method

1. Profile(s) protecting all object layers surrounding the method must allow access;
2. The method's specific profile must allow access; and
3. None of the instance variables accessed by the method must deny access.

Typically, a profile will be associated with the object reflecting the object's overall 'sensitivity'. Profiles will be associated with the methods to implement role-based security: the set of operations any user can perform on an object does not only depend on that user's clearance level, but also on the user's function or role in the organization. Profiles which are associated with instance variables will mainly ~~be~~ used as an additional safeguard to ensure that no method accesses information which the subject should not have access to.

An object, message or variable may have more than one profile associated with it, in which case all the profiles must allow access before access will be granted for a message.

In figure 3 *Object1* is protected by both *Profile1* and *Profile2*, while *Method1* is additionally protected by *Profile3* and *Profile4*. In order to activate *Method1*, permission will have to be obtained from all four profiles; if any one denies access, *Method1* will not be activated.

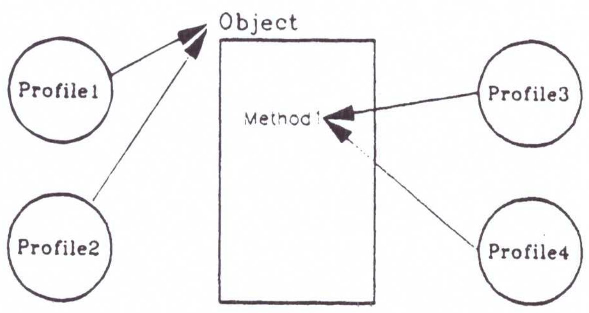

Figure 3: Multiple profiles

4 Gates

4.1 Operation of gates

A gate is a mechanism associated with every object, which will intercept any message leaving the object or arriving at the object. Arriving messages will only be allowed to activate the called method if permission can be obtained from the relevant profile(s). In the case of leaving messages, the profiles associated with the sending object will be tagged to the message. Whenever an instance variable with a profile is accessed, the message will be tagged with that profile. Messages spawned by another message will be tagged with all the profiles associated with the spawning message. If a spawned message returns a value, the spawning message will be tagged with the profiles carried by the returning message; if no values are returned the spawning message will resume execution with all the profiles it had when it spawned the returning message.

All messages crossing the borders between object layers will be checked by the gate, verifying baggage when travelling inwards and attaching profiles when travelling outwards. In order to reference any method or variable at an inner layer, all border crossings between the origin of the reference and the referenced variable or method have to be checked. This ensures that information does not flow to a lower classified class by means of inheritance.

4.2 Examples of message tagging

Assume we are working with a payroll system. All employee salaries are protected by a profile *P*. The payroll object starts sending messages to all employee objects to determine their current salaries. When these messages return the result (the salary) the profile *P* is attached to the returning message. The payroll object inserts the salaries into its internal table—this implies that a variable is being modified, causing the profile *P* to be attached to this variable. Although the payroll object now contains the sensitive salaries, it is not possible to retrieve them illegitemately from this object because they are still protected by their original profile *P*. If this payroll object now sends a message to the cashbook object with this table as a parameter, the profile *P* will again be sent along and (eventually) be attached to the cashbook's variable(s) so that it will still not be possible to obtain the sensitive salaries from the cashbook without the (original) authorization.

Another example of tagging profiles to messages is given in figure 4. There are three objects, *O1*, *O2* and *O3*, six messages, *M1* to *M6*, and eleven profiles, *P1* to *P11*.

If we firstly assume that all messages return values and that the message *M1* is sent to object *O1*, the sequence of events will be as follows:

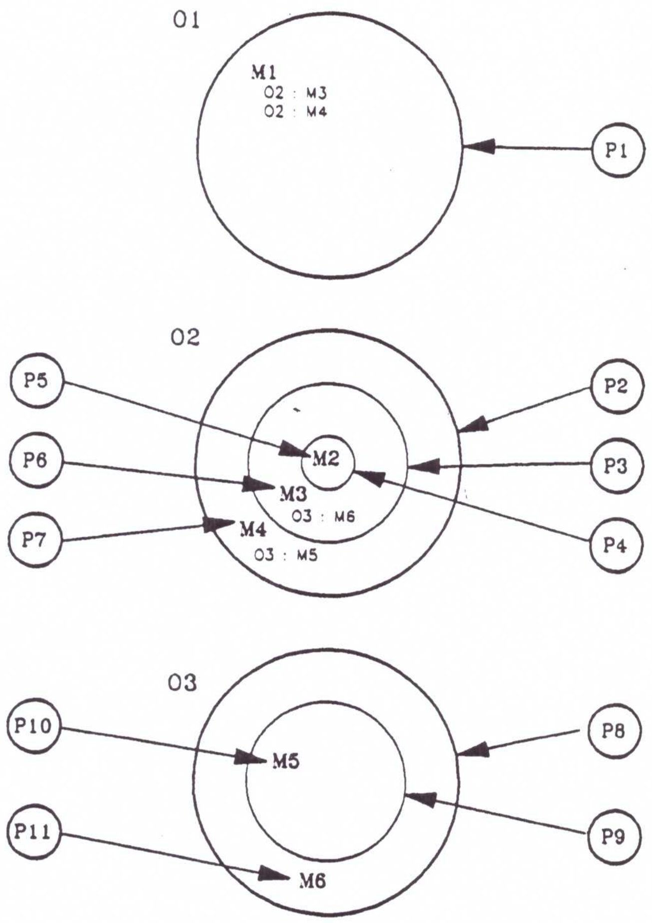

Figure 4: Tagging profiles to messages

Object	Active message	Profiles tagged to message	Remark
O1	M1	-	1
O2	M3	P1	2
O3	M6	P1,P3,P2	3
O2	M3	P1,P3,P2,P8	4
O1	M1	P1,P3,P2,P8,P3,P2	5
O2	M4	P1,P3,P2,P8,P3,P2,P1	
O3	M5	P1,P3,P2,P8,P3,P2,P1,P2	
O2	M4	P1,P3,P2,P8,P3,P2,P1,P2,P9,P8	
O1	M1	P1,P3,P2,P8,P3,P2,P1,P2,P9,P8,P2	

Note the following (numbers refer to the *Remark* column in the table above):

1. We assume that the message starts without any profiles attached to it.
2. Sending the message *M3* to object *O2* causes the activity to 'leave' object *O1* and be transferred to object *O2*. The 'leaving' of *O1* causes *O1*'s profile *P1* to be attached to the message.
3. Object *O2* now send the message *M6* to *O3*. This message first 'leaves' the layer of *O2* which is protected by *P3* and then the layer protected by *P2* causing both *P3* and *P2* to be attached to the message. *P1* is still attached to the message because the message was spawned by *M3* of *O2* which which *P1* was then associated.
4. Control now returns to method *M3* of object *O2*. Since information is returned (our initial assumption), this information 'leaves' the layer protected by profile *P8*, causing *P8* to be attached to the message.
5. Control has now returned from *M3* of *O2*. Again, since the message returned a value to *M1* of *O1*, 'leaving' from *M3* caused profiles *P3* and *P2* to be attached to the message. (Note that *P3* and *P2* are already attached to the message—in an optimized environment it is not necessary to attach them again.)

If we assume, on the other hand, that no messages return values, the sequence of events will be as follows if the message *M1* is sent to object *O1*:

Object	Active message	Profiles tagged to message	Remark
O1	M1	-	
O2	M3	P1	
O3	M6	P1,P3,P2	
O2	M3	P1	1
O1	M1	-	2
O2	M4	P1	
O3	M5	P1,P2	
O2	M4	P1	3
O1	M1	-	4

Note the following (numbers refer to the *Remark* column in the table above):

1. Control returned to method *M3*, object *O2* from method *M6*, object *O3*. Since no value is returned (our assumption) *M3* resumes execution with the profiles it had before sending message *M6*.
2. *M1* resumes execution with the profiles it had before sending message *M3*.
3. *M4* resumes execution with the profiles it had before sending message *M5*.
4. *M1* resumes execution with the profiles it had before sending message *M4*.

4.3 Problems posed by gates

The concept of a gate is the only real deviation in SECDB from the classical object-oriented concept and (not surprisingly) poses the most challenges. These challenges include

1. Handling of messages when access is denied;
2. Cooperation with the object's profile, especially when the profile requires information from the object in order to decide whether access should be granted; and
3. Logical integration into the object-oriented model, especially where such a model is described formally.

Problem (1) can be treated like a normal failure to complete execution of a method, eg when the hardware fails or the disk media is unreadable. This will typically involve abortion of the current transaction and rolling the database back. We do not go into further details here.

Polyinstantiation is often used to solve (1): different 'slots' for a single field are maintained for every clearance level; when a process with clearance level n writes a value to a field, any other process with clearance level n (or higher) can read that value. However, if a process with clearance level m, where m is more trusted than n, writes a value to the same field, all processes at clearance level m and higher will be returned this value when they read the field, while all processes at clearance levels n to $m - 1$ will still be returned the value written by the process at level n. (See [12] for a description of polyinstantiation in object-oriented systems.) Polyinstantiation has to be adapted to be useable in SECDB since data is not necessarily classified at different levels in the PCM model. This means that it is not always possible to determine (and, in any case, it might be dangerous when it is possible) how many 'slots' may exist for the same value. Polyinstantiation will have to be done by attaching 'cascading' profiles to a polyinstantiated object—one level of profile for every copy of the object. If a profile denies access, the profile one level down the cascade is asked for access; this process is continued until a profile allows access; in this case access will be granted to the (copy of the) object on the same level as the granting profile. Only if none of the profiles grant access, will the request be denied. We do not discuss cascaded profiles further in this paper.

Problem (2)—cooperation between an object and its profile—can be illustrated as follows: Suppose some user must have access to every non-manager's employee information. The profile for the employee objects must then first determine whether a specific employee is a manager before granting access. If the fact whether an employee is a manager or not forms part of the employee information, this implies that a profile must sometimes access an object which may not be accessed by the sender of the original message. A similar, but slightly more difficult restriction to solve by ad hoc programming, is a rule allowing access to a salary if and only if the salary is less than \$100 000. In order to handle this kind of restriction, SECDB allows a profile to access an object (by sending messages to it) without getting permission from a profile. To ensure that profiles are not used to extract protected information from an object, access to the profile may be restricted to come from gates (and possibly from some other highly trusted objects, but not from arbitrary objects).

5 Example

The example in figure 5 is intended to illustrate profiles, profile classes and hierarchies of profiles. In a real application we would like to see a classification scheme based on functions (or roles) in the organization, rather than on traditional classification levels. In figure 5 SECURITY_PROFILE is the primitive security profile class. UNCLASSIFIED, RESTRICTED, SYS_ADMIN, etc, are all profile (sub) classes. SECURITY_PROFILE, UNCLASSIFIED, RESTRICTED, CONFIDENTIAL, SECRET and TOP_SECRET form a non-decreasing sequence of profile classes. SYS_ADMIN also offers at least as much protection as SECURITY_PROFILE. However, no such relationship exists necessarily between SYS_ADMIN and UNCLASSIFIED, RESTRICTED, etc. In the example this is used where some profile classes are intended to protect information according to the classification level of the information and another class, SYS_ADMIN, is used to protect system structures, such as profile objects.

EMPLOYEE_CLASS describes a class of employee objects. When defining this class, the system security officer decided to link it to the CONF_PROFILE profile. Every instance of EMPLOYEE_CLASS,

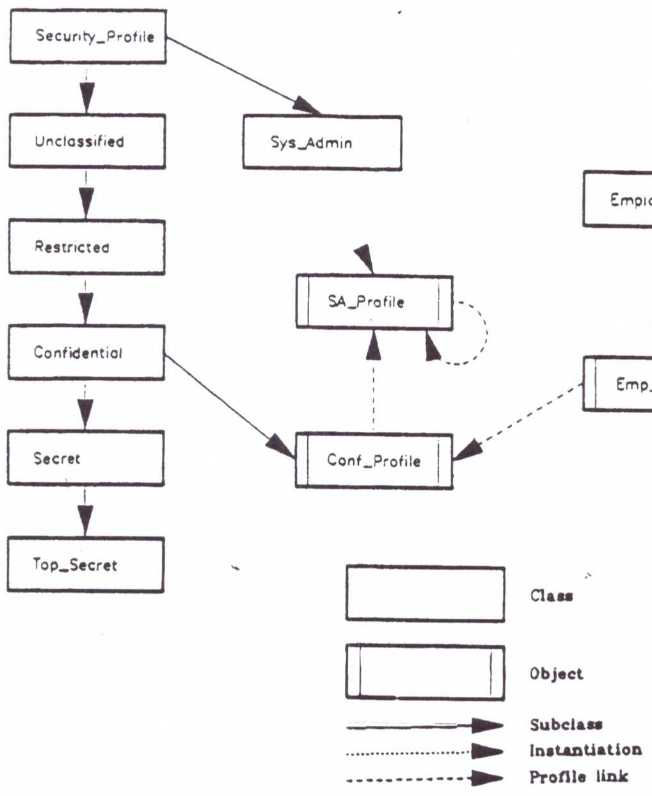

Figure 5: Example

such as EMP_REC, will then be associated with CONF_PROFILE.

6 Model characteristics

In SECDB a previously defined and instantiated security profile is specified (or inherited) for every class whenever such a class is defined.

Subclasses of a class will, by default, inherit the profile of the superclass. A security profile may, however, be specified when defining a subclass, overriding the inherited profile. Viewing an object as a layered entity implies that security (both the simple security property and the *-property) [1] will not be compromised, even if the subclass has less restricting access requirements than its superclass. The simple security property requires that subjects should not have access to information they do not have permission to access and the *-property requires, as mentioned earlier, that no subject can write information where another subject without the proper authorization can then read it.

The simple security property and the *-property have to be demonstrated for two cases:

1. Where messages are sent from one object to another (possibly at different classification levels); and
2. Where an object inherits a message or variable from some superclass (again possibly at a different classification level).

The following arguments do this for the respective cases.

1. When messages are sent from one object to another, the gate/profile combination(s) of every object will ensure the simple security property. Since information sent along with any message carries with it the profile of any object it leaves, the *-property is ensured: any access to that object will have to go through the original containing object's profile, implying that a lower "clearance" level will still not be allowed to access it, although it may now be contained in an object with a lower classification level.
2. When a message or variable is inherited from a superclass, the profile specified in that superclass will be associated with the corresponding layer of the object. Information contained may therefore not be accessed without permission from that security profile, ensuring the simple security property. Also, any information leaving that layer will be tagged with that layer's profile, ensuring the *-property.

7 PCM in the SECDB environment

As we indicated at the start, SECDB has been designed to use PCM. The primary advantage of PCM is its ability to enforce security in potentially non-secure environments, especially in network environments where the identity of the originator of a request is often concealed by layers of application software, network servers and system software. SECDB also shows promise in a distributed database where multiple system security officers may be involved. (The original description of PCM uses the term 'object' to refer to a static entity and the term 'subject' to refer to a dynamic entity; we use the term 'static entity' in this section to avoid confusion in the object-oriented context.)

PCM uses three concepts:

1. A baggage collector to collect 'baggage' about all software and hardware involved in handling a request; baggage is defined as 'the minimum amount of information that has to be collected and must accompany the access request on its route in order that responsibility and access authority checking can be performed even though various transformations or domain crossings may occur';
2. A profile for a static entity which specifies which items are required to take part in handling a request and/or which items are not allowed to take part when accessing this entity; and

3. A validator to compare collected baggage to the accessed static entity's profile.

In SECDB, to fit the object-oriented model, the profile data (containing the allowing and prohibiting contexts) and the validator have been merged into a single profile object.

Although PCM may be used to protect non-object-oriented databases, the object-oriented model is more appropriate, because

- Of the similarity between messages in the two models;
- Of the ability to store data, methods and security information as one coherent unit; and
- In the case of the object-oriented model it is possible to consider security implications at any abstraction level—specifically it is possible to implement role-based security; in the case of other models one can usually only make decisions on whether a subject is allowed to read and/or write specific data records or fields.

In an object-oriented database messages, can originate from objects inside the database or from external sources. Although not the primary goal, SECDB can be used to ensure that requests follow an approved route through the database: some objects are for internal use only, or for use by specific other objects; it is simple to add restrictions to SECDB profiles to ensure that a request only comes via an acceptable predecessor helping to ensure the integrity of the database.

8 Implementation considerations

We do not address the implementation considerations in detail in this paper. However, it must be admitted that a straightforward implementation of the model could be inefficient. Here we point out four areas where optimization shows promise.

Firstly, an object protected by multiple profiles requires extensive checking. If some of the associated profiles have stricter requirements than other associated profiles, the less strict profiles may be ignored. The ordering imposed on profile subclasses stated that a subclass has stricter access requirements than its superclass(es). This means that if a profile from somewhere low in the hierarchy has granted access, the profiles from higher in the hierarchy (instances of ancestor classes) need not be asked for their permission.

Secondly, checks that messages sent inside the environment conform to the profile restrictions can often be done statically (at compile time or system configuration time).

Thirdly, the need for a profile to send a message to its superclass' *GrantAccess* can also be eliminated in some cases: if it is possible to check statically that the access requirements set by a profile class are indeed as least as strict as the requirements set by its superclass.

Lastly, it is unnecessary to tag a profile to a message if the specific profile (or an instance of a decendant class) is already tagged to the message.

It is also worth pointing out that the association of a profile with an object is not expensive: this association can be done by having a pointer (often from the class and not the individual objects) to the relevant profile. This will not require much more memory than a simple sensitivity level would have taken.

9 Comparison with other models

Very few models have been proposed for secure object-oriented databases.

One promising model for a secure object-oriented database system was proposed by Keefe, Tsai and Thuraisingham [7]. Their model, known as SODA, assigns a classification (sensitivity) level to a protected object. Every message which travels through the system, carries with it a current classification level and a clearance level. The current classification level is adjusted whenever an object with a higher classification is accessed. Rules, based on the current classification level and the clearance level and on an object's classification level, determine whether access should be granted to an object.

It can be shown that SODA is similar to a special case of SECDB: define baggage to consist of

1. The clearance level of the sender of a message; and
2. The sensitivity level of any information accessed

and define profiles to contain the sensitivity levels of objects they protect. Note however, that the primary concern in SODA is the protection of variables and objects, whereas methods are the primary concern in SECDB, with variables and objects secondary.

SORION [12] is another multilevel secure object-oriented data model. This model assigns security levels to subjects and entities. A set of rules based on these security levels restricts access.

The access control language of Mizuno and Oldehoeft [10] is based on extended access control lists (ACLs): a four-tuple ACL entry

1. Specifies which user may activate a method;
2. Through which class the request may come;
3. Through which specific object (class instance) the request may come; and
4. Lastly names the protected method.

SECDB is more general; however note that SECDB does not consider classes encountered on the route of the request, but rather objects—ie instances of classes.

Two other methods which may be used to ensure that a request follows an acceptable internal route through the database have been suggested in the literature: views [6] and a law-based approach [9]. In the case of views, an object may define different views—or interfaces—for different objects. A profile in SECDB can specify which objects may access the protected object and also which methods are available to those objects, similar to the view concept. However, this specification in SECDB is not limited to the immediate predecessor object involved in a request. In the law-based approach a set of Prolog rules or laws may be specified to manage the exchange of messages in the system. This is a very general (and powerful) mechanism, enabling (and aimed at) the specification of basic properties, such as inheritance.

10 Further research

Research still remains to be done in the following areas

1. Automating profile generation and automated validation of profiles (for consistency);
2. Identification of unnecessary baggage in order to keep baggage as small as possible;
3. Implications of multiple inheritance on the model; and
4. Designing a notation for specifying security constraints (especially in real-world systems).
5. The model should be applicable to distributed systems where requests may come from many sites, often not even directly connected to the database site. However, the distributed systems may operate concurrently, and the effects of this on the model should be investigated.

11 Conclusion

Object-oriented systems is based on the premise that objects are self-contained entities consisting of data and code. It is meaningful to expect that objects should also have the responsibility to protect themselves. Further, protection in object-oriented systems must fit as cleanly as possible into the generally accepted object-oriented paradigm. In this paper we have proposed a model, SECDB, for supporting security in an object-oriented database, which satisfies these criteria.

SECDB is based on the Path Context Model (PCM), which means that the entire access path is taken into account when it is decided whether a request should be allowed to proceed. This information collected while traversing the access path, is known as baggage. Two concepts are added to the object-oriented paradigm: profiles and gates. Profiles contain criteria for restricting access to an associated object. Gates are mechanisms which will compare baggage accompanying a request to the relevant profile(s), and then either allow or disallow the request to proceed. Profiles are objects themselves. They are also designed to be inherited by a subclass similar to the way instance variables are inherited.

Although SECDB is based on PCM, properties of traditional security models such as the Bell and LaPadula model can still be supported. We illustrated this by indicating how SECDB can emulate SODA, a model for a secure object-oriented database based on the Bell and LaPadula model.

References

[1] DE Bell and LJ LaPadula, "Secure computer system: unified exposition and Multics interpretation", *Rep. ESD-TR-75-306*, March 1976, MITRE Corporation

[2] WH Boshoff and SH von Solms, "A Path Context Model for Addressing Security in Potentially Non-secure Environments", *Computers & Security*, **8**, 1989, 417–425

[3] WH Boshoff and SH von Solms, "Application of a Path Context Approach to Computer Security Fundamentals", *Information Age*, **12**, 2, April 1990, 83–90

[4] WH Boshoff, *A Path Context Model for Computer Security Phenomena in Potentially Non-Secure Environments*, Ph.D Dissertation, Rand Afrikaans University, 1989

[5] A Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983

[6] B Hailpern and H Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE transactions on Software Engineering*, **16**, 11 (November 1990), 1247–1257

[7] TF Keefe, WT Tsai and MB Thuraisingham, "SODA: A Secure Object-oriented Database System", *Computers & Security*, **8** (1989), 517–533

[8] C Laferriere, "A Discussion of Implementation Strategies for Secure Database Management Systems", *Computers & Security*, **9**, 1990, 235–244

[9] NH Minsky and D Rozenshtein, "A Law-Based Approach to Object-Oriented Programming", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1987, 482–493

[10] M Mizuno and AE Oldehoeft, "An Access Control Language for Object-Oriented Programming Systems", *Journal of Systems Software*, **13**, 1990, 3–12

[11] DL Spooner, "The Impact of Inheritance on Security in Object-Oriented Database Systems", pp 141–150 in Landwehr, CE, *Database Security II: Status and Prospects*, North-Holland, 1989

[12] MB Thuraisingham, "Mandatory Security in Object-Oriented Database Systems", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1989, 203–210

[13] APJ Van der Walt, "Random Context Languages Symposium on Formal Languages", Oberwolfach, West Germany, 1970

[14] P Wegner, "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*, **1**, 1 (1990), 7-87