Project IST-1999-11583

Malicious- and Accidental-Fault Tolerance for Internet Applications

SPECIFICATION OF AUTHORISATION SERVICES

Noreddine Abghour
Yves Deswarte
Vincent Nicomette
David Powell

LAAS-CNRS

MAFTIA deliverable D27

Public document

23 JANUARY 2001

LAAS Report 01.001

Table of Contents

Specification of Authorisation Services

Noreddine Abghour
Yves Deswarte
Vincent Nicomette
David Powell
LAAS-CNRS

Abstract: This document describes MAFTIA authorisation services and how they will be implemented in the MAFTIA architecture. The authorisation services implement a fine grain protection, i.e., capable of protecting each object method invocation, in order to satisfy as much as possible the least privilege principle and to obtain the best protection efficacy. The authorisation schemes are flexible and richer than the simple client-server model, thus enabling multi-party transactions to be handled. An analysis of typical scenarios has led to the definition of other functionalities: flexible delegation scheme, separation of duties, degradable capabilities. The authorisation service will be implemented by distributed authorisation servers, which will be made accident- and intrusion-tolerant, and by local security kernels, possibly supported by Java smart cards.

1. Introduction

In computing systems, protection is used for a double purpose:

- *Error confinement*: in case of erroneous execution of a module (procedure, instruction block, object method, etc.), errors can propagate to other modules at some interaction points (procedure calls and returns, branches and jumps, method invocations, etc.). At these points, protection can prevent error propagation if the interaction can be detected as invalid. For instance, memory protection mechanisms should prevent jumps to data segments. So, if they are effective, the protection mechanisms should prevent error propagation and thus provide error confinement in the module where the error occurs.
- *Intrusion prevention*: in the same way, protection mechanisms should prevent non-authorised or invalid actions that could be exploited by malicious users. For instance, memory protection should prevent modification of code segments by non-privileged processes.

As a means for error detection and confinement, protection participates in fault-tolerance. But as a means for intrusion prevention, protection participates also in fault prevention.

Typically, protection is implemented by validity check mechanisms (e.g., checking if an instruction code is valid, or if an instruction format is consistent, or if a memory address is reachable), and by authorisation mechanisms, which checks if each operation is authorised or not, according to some access rights. Both validity checks and authorisation contribute to protection, i.e. to both error detection and intrusion prevention.

Of course, the finer is the protection granularity level, the more efficient is the protection:

- for error detection and confinement, all interactions likely to propagate errors should be checked for validity (and authorisation);
- for intrusion prevention, all actions should be checked for authorisation (and validity).

But also, the finer is the protection granularity, the higher is its cost: more access control information has to be managed, and more time has to be spent for access control. Some trade-off between efficiency and cost must clearly be established.

In the context of MAFTIA, i.e., applications distributed over the Internet, we believe that a good trade-off is to apply protection at the level of object methods:

- this level enables us to cope with suspicious mobile agents, as well as risks of abuse by local (privileged) users;
- the object model is consistent with the current technology for developing distributed applications in heterogeneous environments (Java, ActiveX, etc.);
- this model is richer and more flexible than the usual client-server model, which rules most of the current interactions on the Internet.

Consequently, the authorisation service that we propose for MAFTIA will be able to authorise or deny any object method invocation. As will be shown in this document, the authorisation to invoke an object method will be represented by a capability issued by authorisation servers and checked by local security kernels.

It should also be noted that this approach is more capable of preserving privacy than the usual client-server approach: in the client-server model, the server grants or denies client's access according to the identity of the client, which has to be recorded. Much more personal information is thus recorded than really needed for the transactions. With a more flexible authorisation scheme, the transaction is authorised if the corresponding capability is presented, which discloses no personal information[1]. Only the personal information really necessary for execution of the transaction needs to be disclosed (application of the "need-to-know" principle to privacy).

[1] Of course, generally the identity of the client is known by an authentication server, but, once authenticated, the client can be provided a certificate or a capability which can be used to prove his/her rights without disclosing his/her identity.

This document presents first the global architecture of the authorisation service in Section 2, then describes some typical scenarios to deduce authorisation requirements in Section 3. The functionality of the authorisation services is presented in Section 4, while Section 5 presents some possible extensions. Finally, Section 6 presents the interactions with the other MAFTIA workpackages.

2. Authorisation Architecture

2.1. Authorisation in distributed system

Most authorisation models are based on subjects and objects. A subject is any active entity (e.g., a user, a process, etc) in the system and an object is a passive entity, a data container (typically, a file). All interactions between subjects and objects are controlled by a **reference monitor** [DoD 1985], which checks if each access is authorised or denied.

Protecting a distributed system is a difficult task because there is no global state in such a system. Protecting each site of a distributed system does not correspond to protecting the whole system. The protection of a distributed system is a problem that is directly connected to the notion of trust. What entities can be trusted in the system, and according to this trust, how is it possible to secure the whole system? Furthermore, how, in such an architecture, can the notion of a reference monitor be used?

We present in the following sections two common approaches for protecting distributed systems as well as the drawbacks of these approaches.

2.1.1. Centralised authorisation

This approach is based on a central reference monitor, which checks all accesses from subjects to objects. It requires only one site to be trusted: each access from a subject to an object is handled by this site, which authorises or denies the access. A successful implementation of such a scheme has been proposed with the secure file server of the Newcastle Connection [Rushby & Randell 1983]. In this project, each user at his terminal can access his files through the secure file manager which checks and manages all the file access rights.

One advantage of such an approach is that it allows the security policy of the whole system to be consistent and easy to manage. However, in this approach, the whole system security relies on one machine, which is thus a single point of failure. Moreover, the trusted machine may become a bottleneck and may impede system performance (in particular since even accesses from local subjects to local objects need to be checked through the central reference monitor).

2.1.2. The Red Book approach

In the approach proposed by the “Red Book” [NCSC 1987], each site of the system has a Trusted Computing Base (TCB) and can be evaluated according to the NCSC criteria of the “Orange Book” [DoD 1985].

Each TCB contains a reference monitor which checks not only all local accesses but also all accesses from remote subjects to local objects and all accesses from local subjects to remote objects. Moreover, in this approach, each TCB trusts all the other TCBs of the whole system. More precisely, each TCB is confident that the other TCBs will prevent:

- that a remote subject would be able to impersonate another remote subject to access genuine local objects;
- that a genuine local subject would be denied a legitimate access to a remote object, or be granted an illegitimate access to a remote object.

This set of TCBs is called a NTCB (*Network Trusted Computing Base).*

Although there is no bottleneck in the Red Book approach, we think that the consistency of the security policy is much more difficult to maintain than in the centralised approach because, in order to carry out access control, each TCB must know all[2] the subjects of the system (local and remote) and their rights to access local objects. Furthermore, if any of the TCBs of the system is compromised, the security of the whole system is compromised, due to TCB mutual trust.

We propose and describe in Section 4 a protection scheme for distributed systems that does not have any of the drawbacks of these two approaches.

2.2. Protection of object-oriented distributed systems

One of our goals is to build an authorisation scheme that fits the object-oriented programming paradigm. First of all, an object, as used in this paradigm, is made up of private state information and a set of operations that represent the object’s interface. The state of each object is represented as a set of variables. The operations defined on objects are called **methods** and are the only way to modify the state of the object or to get information on this state. The invocation of a method leads to the execution of the corresponding operation. As regards protection, this notion of object obviously covers both active and passive entities (which usually define the subject and object notions). Furthermore, all accesses to objects are made by means of messages, which provoke information flows. Throughout the rest of this report, we will only consider objects, as being both active and passive entities. Thus, the only access rights to be considered in our system are the rights for an object to invoke other objects’ methods.

A distributed object system is composed of many objects, which may be of different kinds. Some objects are coarse-grained persistent objects. Others are transient objects that only exist in order to momentarily fulfil a part of an operation in the system. It is important to separate the protection of these two sets of objects. Furthermore, most of the operations executed in such a system are *high-level operations,* i.e., operations whose realisation involves the collaboration of several objects on different sites. For example, sending mail to a user involves at least the collaboration of mail transport agents, a mail delivery agent, a file server, a file and a shell. We propose in this report an authorisation scheme that can handle the management of access rights for such high level operations.

[2] Or at least all the subjects which can access local objects.

2.3. Two levels of protection

The authorisation scheme we propose distinguishes two levels of protection. The protection of coarse-grained persistent objects of the system is realised by a distributed **authorisation server**. This authorisation server is able to manage access rights for high-level operations in the system (see Section 4.2). A second level of protection is implemented on each site of the system: a **security kernel** is responsible for checking all accesses to local objects and furthermore, is responsible for locally managing access rights for all transient local objects.

2.3.1. The authorisation server

In the usual client-server model, the authorisation is enforced by the server: the server decides to fulfil or deny the client request, according to the client's identity (verified by some authentication scheme) and according to some locally-enforced rules. When more than two entities are involved, the client can delegate some of its rights to a server, which can then act on behalf of the client in requesting a service from another server. This can be done by using a proxy, such as in Kerberos V5 [Kohl & Neuman 1993], SESAME [Parker 1991] or CORBA [OMG 1995].

This scheme presents several drawbacks. First, the delegate has to be trusted by the client: the delegate is authorised to use (and possibly abuse) the client's privileges to perform actions, usually even with the client's identity. If malicious, the delegate can perform actions unwanted by the client. Moreover, he may usually do so with impunity since these actions will be attributed to the client. The second drawback is that the client must possess more privileges than necessary, in order to be able to delegate these privileges (see Section 4.1). A third drawback is that the server has too much responsibility: since the server is the only entity that enforces the authorisation, this scheme is ill adapted to peer-to-peer communications or other transactions involving several mutually-suspicious entities. Some of the drawbacks have been addressed by some proxy implementations [Gasser *et al.* 1989] or by more sophisticated authorisation schemes such as [Wulf *et al.* 1974, Biskup & Brüggemann 1988, Nicomette & Deswarte 1997].

The authorisation scheme we propose is one of these sophisticated authorisation schemes and implements corresponding servers, which are third-party entities able to manage various flexible authorisation schemes. An authorisation server has the responsibility to grant or deny authorisations for operations that will be executed by the various parties involved. More precisely, in our scheme, an authorisation server is responsible for managing access rights for persistent objects of the system. A server decides, for each access to a persistent object in the system, if this access is to be authorised or denied. In order to take these decisions, the authorisation server stores an access matrix (in which are defined the rights to access persistent objects) and manages rules that allow **capabilities** and **vouchers** (see Section 4.2) to be built and delivered to the objects that are authorised to access persistent objects. As shown in Section 4.2.5, our delegation scheme does not have the drawbacks presented above and it strictly obeys the least privilege principle.

The notion of authorisation servers is not new, and authorisation servers implementing simple schemes have already been proposed, for instance, in Delta-4 [Blain & Deswarte 1990], HP

Praesidium [HP 1998] or Adage [Zurko *et al.* 1999]. However, in addition to implementing more comprehensive authorisation schemes, *MAFTIA* authorisation servers will be trustworthily tolerant to intrusions (see Section 2.4.1).

2.3.2. The security kernels

Each site of the system has a security kernel. This security kernel plays two main roles:

- it checks all accesses to local objects, whether persistent or transient;
- it autonomously manages all access rights for local transient objects.

The security kernel controls all accesses to local objects by checking if each request carries a capability that authorises the access. This capability may have been delivered by the authorisation server, if the access is an access to a persistent object, or by the security kernel itself, if the access is an access to a local transient object.

Each security kernel must enforce the usual properties of a reference monitor, as defined in the Orange Book, i.e., each security kernel must be tamperproof, must always be invoked and must be small enough to be subjected to exhaustive analysis and tests, the completeness of which can be assured. We thus trust each security kernel up to a certain extent: we consider that it is very difficult to control any of the security kernels, but even if an intruder succeeds in controlling a particular security kernel, the security of the whole system is not compromised. As a matter of fact, this attack means that the intruder is able to control all accesses to local objects but cannot be granted access to remote objects, or impersonate a fake object for remote operations.

In fact, the security kernel of each site does not trust the other security kernels of the system; it only trusts the authorisation server. This means that this approach does not have the drawbacks of the Red Book approach.

2.4. Application to MAFTIA

2.4.1. A fault and intrusion tolerant authorisation server

The authorisation server presented in Section 2.3.1 can be made fault-tolerant and intrusion-tolerant [Deswarte *et al.* 1991, Reiter *et al.* 1994]. Various techniques may be used, such as for example, the **Fragmentation-Redundancy-Scattering** technique [Deswarte *et al.* 1991].

Fragmentation-Redundancy-Scattering (FRS) is a method for tolerating accidental faults and malicious intrusions, preserving confidentiality, integrity and availability. Fragmentation consists of splitting information into different fragments so that once isolated, every fragment contains no significant information. Redundancy is applied to the fragments (e.g., by replication) so that modification or destruction of a small number of fragments will not prevent the reconstruction of the information. Scattering refers to the way by which each fragment is isolated from the others. Various kinds of scattering can be applied: topological scattering, temporal scattering, frequency scattering, privilege scattering [Deswarte *et al.* 1991].

Once this operation has been carried out, an intrusion into a part of the system gives access only to isolated fragments, and thus to non-significant information. The number of intrusions that can be tolerated without delivery of significant information depends on the way the operations are implemented. In the same way as in classic fault tolerance, the number of faults that can be tolerated depends of the level of redundancy and of the kind of faults. This number is a parameter that can be chosen with regard to the trade-off between security, cost and performance.

To apply the FRS technique, the authorisation server has to be composed of several *security sites*, and only a *majority* of these security sites has to be trusted. This means that the failure of, or successful intrusion into, a minority of the security sites does not compromise the security of the whole system: all unclassified data are replicated on the different security sites, the confidential data are fragmented and scattered on the security sites (e.g., using threshold cryptography) and all security sites perform a majority vote for each decision of the authorisation server. The security sites are administered by different persons, so that a collusion of a majority of these administrators is needed for the authorisation server to perform an illegitimate operation.

The FRS technique allows the authorisation server to be tolerant to intrusions such as:

- somebody outside the system who gains access to one site;
- a legitimate user of the system who tries to access information or services for which he has no right;
- a security administrator who abuses his privileges to perform illegitimate actions.

The following figure presents the distributed authorisation architecture.

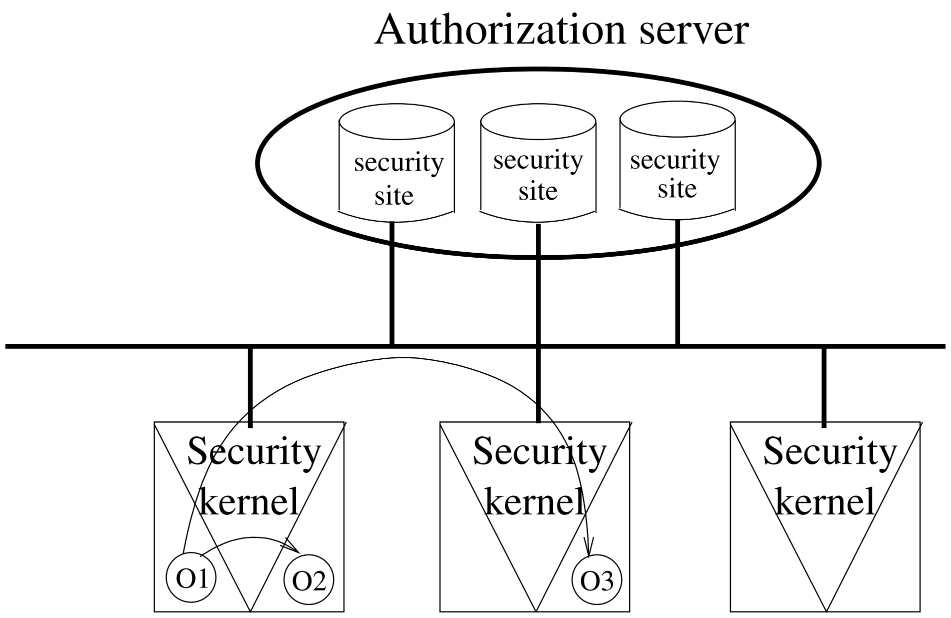

Figure 1: Distributed authorisation architecture

2.4.2. Security kernels in a wide-area setting

The security kernels have already been experimented in the context of local area networks [Nicomette 1996]. In such an environment, the implementation of the security kernel on each site of the system is rather simple, since the different user sites can be considered as homogeneous and under the control of a single authority: the same security kernel software can be locally installed on each site of the network. So only one security kernel has to be developed, and each site can be trusted to run a copy of the security kernel.

But in the context of wide-area networks (such as the Internet), the implementation of such a security kernel is more complex, due to the heterogeneity of connected sites: it would be necessary to develop one version of the security kernel for each kind of site; and since the sites are not under the control of a global authority, there is no way to ensure that each site is running a genuine security kernel, or the same version thereof.

Nevertheless, we have to trust something on each site of the distributed system in order to make some controls. We can consider the following solutions:

- The security kernel can be implemented by using the protection mechanisms of the Java Virtual Machine such as the Sandbox mechanism, or the access control list mechanism of the JDK 1.2.
- The implementation of security policies by means of the reflection technique as in [Welch & Stroud 2000]. Thanks to this technique, the execution of an object can be intercepted at load-time in such a way that access control can be executed by non-functional code independent from the code of the object itself.
- The in-lined reference monitor enforcement of Java Stack Inspection as proposed in [Erlingsson & Schneider 2000]. This method defines how an in-lined reference monitor can be merged into Java applications to enforce security policies such as stack inspection. Briefly, this technique consists in merging at load-time checking code into the application itself. The application is thus transformed into a secure application that is guaranteed not to take steps violating the security policy being enforced.
- Java Card technology, which seems to us the best implementation of security kernels. Code embedded in a Java card allows automatic capability verifications to be done easily. Furthermore, since a Java Card may possess its own private key, it is possible to generate capabilities for a specific Java Card (i.e., for a particular security kernel). This is probably the easiest and most secure way to implement a security kernel in a totally non-homogeneous environment. Furthermore, this implementation is consistent with our object model.

It must be noted that, except in the case of the in-lined reference monitor above, these solutions require to manage security kernel versions consistently over the distributed system so that an obsolete security kernel does not impede the authorisation efficiency.

2.5. Summary

Thanks to this collaboration between the authorisation server and the different security kernels, we can protect a whole distributed application without the risk linked to the existence of a single point of failure (we apply fault-tolerant techniques to the authorisation server to eliminate this weakness) and also without excessive confidence in each site as requested by the Red Book approach.

3. Some typical scenarios and their requirements

In this section, we examine two important Internet applications that require authorisation. For each application, we present the basic functionality, including some attacks that can threaten the application, and we then deduce the authorisation requirements for the applications.

3.1. French healthcare network (simplified sketch)

From the beginning of the seventies, the French health insurance establishment (CNAM: Caisse Nationale d'Assurance Maladie) has been seeking to modernise its system of reimbursement of medical expenses to social security beneficiaries, in order to improve the management of health care expenses and reduce the cost of reimbursement processing. To achieve these objectives, the national health insurance is planning to replace the traditional paper-based healthcare forms by electronic healthcare forms.

Like its ancestor paper form, the electronic healthcare form gathers all information necessary for the reimbursement to the beneficiary, this information being protected by ciphering and by cryptographic signatures using keys stored on dedicated smart cards. These electronic forms are transmitted by the healthcare professionals over a digital network to CNAM and other involved health insurance companies and organisations.

In the following section, we will describe the participants and transaction flows of the French healthcare system, and then present some attacks that can occur during the processing of the medical forms. We then deduce some authorisation requirements.

3.1.1. Identification of components

The French healthcare system uses two kinds of smart cards:

- The *CPS card* (Carte de Professionnel de Santé, health professional card, <http://www.gip-cps.fr/>): is a smart card identifying the healthcare professional (general practitioner, specialist, dentist, laboratory biologist, nurse, pharmacist, physiotherapist, etc.). Besides medical treatment, it allows the health professional to issue prescriptions, and to encrypt and sign electronic healthcare forms.
- The *Vitale card* (carte Vitale, <http://www.sesam-vitale.fr/>): is a patient smart card containing administrative and treatment data. It serves as a proof of social security

affiliation, and allows the identification of the patient as well as the immediate recognition of the rights of the insured person by indicating his/her affiliation to insurance organisations.

An *electronic healthcare form* is a file containing the same data that was previously held on a paper-based healthcare form. It serves as a proof of a medical act carried out or prescribed by the healthcare professional.

Once a day, the electronic healthcare forms are sent by the health professional to a *frontal system*. The frontal systems are routers run by private organisations, which receive the batches of electronic healthcare forms from the healthcare professionals and dispatch them to the concerned insurance organisations or companies.

3.1.2. Identification of relationships

3.1.2.1 Doctor-patient: creation of the electronic healthcare form

Typically, at the beginning of the day the healthcare professional introduces his[3] CPS smart-card in the computer reader and enters his personal code. The CPS card is used to control access to patient records or other medical databases located on the computer.

When the patient requests treatment from the healthcare professional, she must present her Vitale smart card that contains the data concerning social security affiliation. This is also a way to enforce the identification of the patient. The healthcare professional after a consultation or other medical act, by using his CPS card and the patient’s Vitale smart card creates an electronic healthcare form, on which he completes administrative information about the patient, the required medical act, and detailed cost of the treatment provided. After completion, he stores a copy in the patient’s Vitale smart card and saves another one in his own computer hard disk for transmission to the healthcare insurance for the reimbursement. In addition, he gives the patient her paper prescription.

3.1.2.2 Pharmacist –patient

To obtain medications prescribed by the doctor, the patient must give the pharmacist of her choice the paper prescription and present her Vitale card. In the same way as the doctor, the pharmacist uses both his own CPS smart card and the patient’s Vitale smart card to print a paper healthcare form on which he sticks the medication “proof-of-purchase” labels. Later, he sends this paper form to the patient’s health insurance organisation for the reimbursement to be carried out.

[3] To avoid a cumbersome “he/she” formulation, throughout this document we will use “he” for the doctor, the pharmacist and the insurance operator and “she” for the patient and the secretary. No sexism is implied!

3.1.2.3 Doctor-secretary

The transmission of electronic healthcare forms is considered as an administrative task. So, typically, at the end of each day the healthcare professional delegates the responsibility to accomplish this mission to his secretary, by lending his own CPS card. The secretary uses the CPS card to encrypt and sign the batch of electronic healthcare forms created in the day. The signature certifies both the benefits and the payment of the medical act that opens the right to reimbursement. Then the secretary sends the whole batch to the frontal system, which according to various criteria routes the healthcare forms to the appropriate insurance processing centres.

3.1.3. Examples of possible attacks

Medical treatment requires a trust relationship between patient and healthcare professionals. The privacy of their relationship should be protected against third party interests (e.g., private insurance companies); medical diagnoses and sensitive data should be kept strictly confidential. This specific rule would not be respected in the current system, when the healthcare professional delegates to his secretary the responsibility for transmitting electronic healthcare forms, since the secretary can use the CPS card to access to all sensitive information about patients. Moreover the secretary can collude with a malicious patient, and alter or create fake electronic healthcare forms. As a matter of fact, by transmitting his CPS card to his secretary, the healthcare professional transfers her all his rights, i.e., read/write access rights to patient records, creation rights on electronic healthcare forms. This is due to the kind of protection currently used for electronic healthcare forms, based only on ciphering and digital signatures by means of keys stored on the CPS card. Thus, to encrypt, sign and transmit the electronic form batch, the secretary needs the doctor’s CPS card. But with the doctor’s CPS card, she has all the doctor’s rights, and thus can also create, modify or delete patient records and electronic healthcare forms. A more flexible delegation scheme is clearly needed.

We can envisage similar attacks on confidentiality within the health insurance processing centre when the operator is verifying electronic healthcare forms. It may be that the operator does not just verify that the prescribed medications are those to be reimbursed, but he could also access the patient’s identity and other personal health information, which is not needed for the verification. In theory, the operators should not have access to this type of information and the verification should be done anonymously, just by checking the consistency between electronic and paper forms. Today, the privacy of a patient’s health information is not covered by the current system, because electronic healthcare forms are just protected by cryptographic mechanisms that authorise or completely deny access to all the information. Clearly, a fine-grained authorisation scheme is needed.

3.1.4. Requirements

From the situations above, we point out two important requirements that authorisation schemes must satisfy to keep sensitive information from being disclosed to unauthorised recipients, and to ensure that information is created and modified only in a specified and authorised manner: a general authorisation scheme should be able to ensure separation of duties and delegation of fine-grained access rights.

3.1.4.1 Separations of duties

In some circumstances, adequate separation of duties has a major impact on ensuring that operations are valid and properly recorded. By the term “separation of duties” we mean that some operations require the cooperation of different persons, with different roles or duties. No single person should be able to execute completely such an operation from initialisation to completion. In the examples above, a healthcare professional is not capable of creating any electronic healthcare form without collaboration of the patient. This separation of duty is currently realised through the use of smart cards: both the CPS and the Vitale cards must be inserted in the corresponding card readers for the computer to be able to create an electronic healthcare form. This mechanism is sound, based on strong cryptography. But it is not flexible enough to enable fine-grain authorisation on various operations. A more flexible mechanism should be implemented in a general authorisation scheme.

3.1.4.2 Delegation of fine-grained access rights

Delegating fine-grained access rights is a way to enforce the least-privilege principle. It requires that a user be given no more privileges than necessary to perform an operation. Ensuring fine-grained delegation requires identifying what is the operation to be delegated, determining the minimum set of privileges required to perform that operation, and distributing to the delegate those privileges and nothing more. In the above example, when the doctor lends the secretary his own CPS card to transmit electronic healthcare forms he must be able to grant the secretary the least privilege required to accomplish this task, and nothing more. For example, giving her the right to encrypt, sign and send a batch of electronic healthcare forms without the rights to read or edit them.

3.2. Web auction

In the past few years, web auctions have grown from nothing to an activity measured in hundreds of millions of dollars. Compared to projections of various forms of electronic commerce, auctions have been among the most successful.

In an auction, goods can be sold at a price determined through interactions between a seller and several possible buyers. The seller must have a product that he is willing to sell for the highest price offered. This type of auction can be decomposed into two stages. In the first stage, the seller advertises his product by publishing the product description. In the second stage, the customers bid for the product announcing the bid price $x (meaning “I am willing to buy at the current price bid $x”). The auction is bounded by a deadline known by all participants. In most cases, the bidders can withdraw their bid at any time before the end of the auction. At the end of the auction, the highest bidder acquires the goods by paying the price he bade.

Our study of the web auction is not motivated by their practical importance, but by some novel security problems that they raise.

3.2.1. Example of possible attacks[4]

The basics of a web auction are fairly simple and organised as a succession of rounds of bidding. Before the opening of the bidding the seller publishes the product description, and customers have a limited time to make an offer. This sort of auction has some weakness. In particular, when an intruder places a bid price much higher than what the product is worth, with the intention to discourage other bidders from placing higher bids.

Consider a situation where first bidder N°1 bids $10 for the product. Another bidder N°2 comes along and he wants the product for $20, so he bids that amount. So far the auction goes on normally. At the third step an intruder bids an extremely high amount $10000, to prevent other bidders from placing higher bids. From this round, all other bidders leave the auction because the current higher bid is much higher than the product’s real value. But just a few seconds before the auction closes, the intruder cancels his previous bid price ($10000) and bids only $21, allowing him to win the auction as the bidder with the highest offer.

To cope with this problem, and to make the auction usable, we must supply a mechanism that prevents the bidders from placing a lower bid than the one they sent previously.

3.2.2. Requirements: Degradable access rights

The mechanism we propose to cope with this problem is that of degradable access rights. A degradable access right is a capability that can be used several times, but with a lower privilege at each use. In the above example, each bidder can receive a capability for unlimited bids. But once the capability has been used (i.e., a bid has been placed), the same capability is only valid with higher constraints: in this case, the capability is valid only for bids higher than those placed previously. This solution is more efficient than requesting a capability for each bid, which would require a communication with the authorisation server each time.

This kind of degradable capability is a facility that can probably have many applications. On the contrary, it does not seem reasonable to consider upgradeable capabilities.

[4] Auction web sites have suffered many attacks and other malfunctions, including those described in this section. See for instance:
http://news.cnet.com/news/0-1007-200-334943.html?st.ne.fd.gif.d
http://uweb.superlink.net/jason/ebay/,
http://catless.ncl.ac.uk/Risks/20.51.html#subj3
http://cgi.ebay.com/aw-cgi/eBayISAPI.dll?RetractBidShow

4. Authorisation Services

This section describes the services provided by the authorisation servers.

4.1. The least privilege principle and the delegation problem

Protection efficiency relies mostly on how well the least privilege principle is implemented. The authorisation scheme we propose aims at satisfying as much as possible this principle. When a high-level operation is executed in the system (involving cooperation of several objects), we want each object to execute its part of the high-level operation with the privileges just necessary for that task but no more. We want to control the propagation of rights inside the system to obey the least privilege principle as much as possible. In most systems, this is not realised; for instance:

- A Unix process always runs with all the privileges of its user while, in most cases, it does not need all these privileges. Moreover, by means of the SUID bit, a user can impersonate another user for executing a specific operation. For instance, a user who changes his password runs a process with all root privileges while he needs only to update data in `/etc/passwd` (or `/etc/shadow`).
- In some object-oriented database systems [Ahad *et al.* 1992, Bertino 1992, Richardson *et al.* 1992], each access in the system is realised through the invocation of a method of an object. Fined-grained access controls are thus realised, which is advantageous for respecting the least privilege principle. But, in these systems, all access controls are always realised according to the identity of the user who has generated the request. An object that realises an operation is never itself considered as a principal in the system. Each elementary invocation method is thus always realised with all the privileges of the user who runs a global task in the system.
- The usual notion of privilege delegation found in most distributed systems (for example DSSA [Gasser & McDermott 1990], Kerberos [Kohl & Neuman 1993], SESAME [Parker 1991]) does not strictly enforce the least privilege principle either. In these approaches, when a subject s delegates privileges to another subject s', the operations performed by s' are checked according to the identity of s. This means that s has to possess the corresponding rights even though it does not use them. For example, to print a file, a client must also possess the read right on this file: for a file to be printed, the print spooler has to read the file before sending it to the printer. The client has thus to possess the read right on the file to be able to transfer this right to the print spooler. This means that in order to print a file, a user must first possess the right to read the file, which is much more powerful than just the right to print: with a read right the user can create copies of the file and distribute these copies to other users, etc.

We think that to strictly implement the least privilege principle, a subject must be able to transmit privileges that it does not even own. Thus, the access controls must be made according to the identity of the subject that realises the access and not according to the identity of the object that has transmitted the privilege. The authorisation scheme we present in this report uses a new scheme of privilege delegation to strictly implement the least privilege principle.

4.2. Activities, symbolic rights and vouchers

In our authorisation scheme, we distinguish persistent object protection, managed at the authorisation server level, and local transient object protection, managed at the security kernel level.

Access right management for transient objects is very simple: it is just the responsibility of each security kernel to create an owner capability for the transient object, and deliver this owner capability to the creator object. With this owner capability, the owner can ask the security kernel to create capabilities for each method of the local transient object and, discretionarily, can transmit these capabilities to other local objects.

Regarding access right management at the authorisation server level, our goal is to define an approach that allows privileges to be built and transmitted for high-level operations in the system in a quasi-automatic way. The key point of this approach is its simplicity and its flexibility.

Our model is based on the notions of **activity**, **symbolic rights**, **capabilities** and **vouchers**. We detail these notions in the following sections.

4.2.1. Activity

An object may invoke a method of another object on its own, merely to perform some precise operation that it needs. This kind of access corresponds to an elementary operation. On the other hand, there are also some invocations that are made only because they are involved in the realisation of a high-level operation in the system. For instance, the sending of mail from a user *u* to a user *u'* is a high-level operation that involves the cooperation of several objects: the mail transport agent, the mail delivery agent, the mailbox of user *u'*, etc. Such a high-level operation is called an **activity** in our model. An activity is composed of a set of method executions on different objects in the system. These method executions are functionally linked together and cooperate in the same goal. From the security management point of view, it is easier and more natural to distribute rights to execute high-level operations rather than to grant users all the rights needed for the execution of all the methods involved in the corresponding activity. We will show that this approach also contributes to satisfying the least privilege principle.

4.2.2. Symbolic rights

As we distinguish two types of operations in the system, we also distinguish two types of access rights linked to these operations: the **method rights** and the **symbolic rights**. A method right is a right to invoke a particular method of a particular object. A symbolic right is a high level right that is used in order to authorise the realisation of a high level operation in the system (e.g., the right to print file *f* on any printer of the system is a symbolic right). In general, several symbolic rights are needed to authorise the realisation of a single high-level operation.

Method rights and symbolic rights are stored in the access control matrix (managed by the authorisation server). In the matrix, the rows and the columns represent users, roles, classes or objects.

A **role** represents a set of users having the same duty (e.g., Administrators). In fact, a role is a user attribute and a single user may have several roles corresponding to different duties. Roles are used as matrix entries instead of user identifiers to reduce the size of the access control matrix and to simplify its management. Indeed, the rights to access an object or a class are often defined according to duties (i.e., roles) rather than for single users of the system. Roles are a flexible way to gather and to manage users. In the same way, the notion of **class** helps to define rights to access a set of objects (all the instances of the class) and allows the access matrix management to be kept simple.

In the access control matrix, the access rights that an entity *e* holds for another entity *e'* can be found at the intersection of the row *e* and the column *e'*. The two access right families (method rights and symbolic rights) are expressed in two different ways:

- the **method rights** that an entity *e* holds for an entity *e'* are expressed by the list of the corresponding methods of *e*. For instance, if M(*e*,*e'*) = *(f,g)* then the entity *e* has the right to invoke methods *f* and *g* of *e'* and is not authorised to invoke any other method of *e'*
- the **symbolic rights** are briefly described in the next Section.

4.2.3. Symbolic right rules

A symbolic right rule describes the set of all symbolic rights that must be found in the access matrix in order to authorise the realisation of a high-level operation.

4.2.3.1 Definition

Let `M` be the access matrix, `U` the set of all the users in the system, `C` the set of all the classes in the system, `O` the set of all objects (instances) in the system, `R` the set of all roles in the system, `Role(u)` the set of roles of the user `u`, `Class(o)` the class of the object `o`.

`SubClass(X)` is a set of classes composed of class `X` and its descendants, i.e.,

$$\texttt{SubClass(X)} = \{\texttt{X}\} \cup \{\texttt{Z, Z is descendant of X}\}$$

Symbolic rights are represented as terms: $\texttt{sr}(\texttt{arg}_1, \texttt{arg}_2, \ldots, \texttt{arg}_n)$ where `sr` refers to a type of action in the system and $\texttt{arg}_1, \texttt{arg}_2, \ldots, \texttt{arg}_n$ are objects, classes, users or roles (i.e., $\forall \texttt{i}, \texttt{arg}_\texttt{i} \in \texttt{C} \cup \texttt{O} \cup \texttt{U} \cup \texttt{R}$). For instance, `SendMail(FILE,john)` represents the right to send a mail to user john from any file of the system. A symbolic right may be only found in the access matrix in a column which corresponds to one of its arguments. If the symbolic right is stored in the column of the argument $\texttt{arg}_\texttt{i}$, then the symbolic right will be noted $\texttt{sr}(\texttt{arg}_1, \ldots, \texttt{arg}_{\texttt{i-1}}, \texttt{this}, \texttt{arg}_{\texttt{i+1}}, \ldots, \texttt{arg}_\texttt{n})$. For instance, the symbolic right `SendMail(FILE,john)` in the column of class `FILE` is noted `SendMail(this,john)`. The line where the symbolic right is stored represents the entity the symbolic right is granted to (i.e., a user, a role, an object or a class).

A high level operation in the system is represented as: **op**(**arg**$_1$, …, **arg**$_n$). **A**(**op**) (**e**,**arg**$_1$, …, **arg**$_n$) represents the authorisation to perform the high level operation for the entity `e` (`e` may be a user, a role, an object or a class).

A symbolic right rule describes the set of all symbolic rights that must be found in the access matrix in order to authorise the realisation of a high level operation. We express each symbolic right rule according to the following formalism:

$$\frac{\texttt{precond}}{\textbf{postcond}}$$

where `precond` is a set of symbolic rights and **postcond** represents the authorisation to realise a high level action. More precisely, each rule $R(e,op,sr,arg_1, \ldots, arg_n)$ is defined as follows:

$$R(e,op,sr,arg_1, \ldots, arg_n):$$
$$\frac{sr_{e,arg_1}(arg_1, \ldots, arg_n), \ldots, sr_{e,arg_n}(arg_1, \ldots, arg_n)}{\mathbf{A(op)(e, arg_1, \ldots, arg_n)}}$$

All the symbolic rights that compose the precondition are evaluated according to the evaluation function $eval_M$ on matrix M. This evaluation function is defined in the following way:

$$eval_M(sr_{e,argi}(arg_1, \ldots, arg_n)) = TRUE$$
$$\Leftrightarrow \exists\, \alpha_1, \ldots \alpha_{i-1}, \alpha_{i+1}, \ldots \alpha_n \in C \cup O \cup U \cup R\ ,\ \alpha_k \mathrel{\tilde{\supseteq}} arg_k \quad \forall k \in 1, \ldots, i-1, i+1, \ldots, n$$
$$\text{and } M(e,arg_i) = sr(\alpha_1, \ldots, \alpha_{i-1}, this, \alpha_{i+1}, \ldots \alpha_n)$$

The relation $\tilde{\supseteq}$ ($\tilde{\supseteq} \subseteq (C \cup O \cup U \cup R)^2$) is defined as follows:

$$a \mathrel{\tilde{\supseteq}} b \Leftrightarrow \begin{cases} a=b & \text{if } a,b \in O \\ Class(b) \in Subclass(a) & \text{if } a \in C, b \in O \\ a=b & \text{if } a,b \in U \\ b \in Role(a) & \text{if } a \in U, b \in R \end{cases}$$

A symbolic right with users or roles as parameters is a relatively new notion. An entity is granted a symbolic right which applies to a user when some objects owned by this user will have to be read or written during the execution of the high level task. This symbolic right will allow method rights for one of the user's files for example to be built and granted during the execution of the high level task.

Either an object or a user may ask to realise a high level operation:

- If a user u wants to carry out the high-level operation **op(arg**$_1$**, …, arg**$_n$**)**, then the system checks the rules $R(u,op,sr,arg_1, \ldots, arg_n)$ but also $R(r,op,sr,arg_1, \ldots, arg_n), \forall r \in Role(u)$. If the evaluation of the precondition of one of these rules is "TRUE", then the user is authorised to carry out the operation.
- If an object o wants to carry out the high-level operation **op(arg**$_1$**, …, arg**$_n$**)**, then the system checks the rules $R(o,op,sr,arg_1, \ldots, arg_n)$ but also $R(c,op,sr,arg_1, \ldots, arg_n), \forall c, Class(o) \in Subclass(c)$.

4.2.3.2 Example

Let us define the following symbolic rule:

$$\frac{\texttt{C1,C2,C3}}{\mathbf{\texttt{A(recordscene)(e,rec,cam,tape)}}}$$

With:

$\texttt{C1} = \texttt{RS}_{\texttt{e,rec}}\texttt{(rec,cam,tape)}$
$\texttt{C2} = \texttt{RS}_{\texttt{e,cam}}\texttt{(rec,cam,tape)}$
$\texttt{C3} = \texttt{RS}_{\texttt{e,tape}}\texttt{(rec,cam,tape)}$

`A(recordscene)(e,rec,cam,tape)` represents the authorisation for entity `e` to realise the high level operation "recording a scene from the movie-camera `cam` with the recorder `rec` on the video tape `tape`". The entity `e` is authorised to realise this operation only if the following symbolic rights can be found in the access matrix for `e`:

- the right to record a scene with the recorder `rec` for a set of movie-cameras that includes `cam` and for a set of tape that includes `tape` ($\texttt{RS}_{\texttt{e,rec}}\texttt{(rec,cam,tape)}$)
- the right to record a scene from the movie-camera `cam` for a set of recorders that includes `rec` and for a set of tapes that includes `tape` ($\texttt{RS}_{\texttt{e,cam}}\texttt{(rec,cam,tape)}$)
- the right to record a scene on the tape `tape` for a set of recorders that includes `rec` and for a set of movie-cameras that includes `cam` ($\texttt{RS}_{\texttt{e,tape}}\texttt{(rec,cam,tape)}$).

Two examples of such symbolic rights for a user `u` and a role `director` in the access matrix are presented below (`RECORDER`, `CAMERA` and `TAPE` are classes; **`Rec`**, **`Cam`**, and **`Tape`** are objects, instances of these classes; the symbol `this` represents the entity corresponding to the column in which the right is placed).

	Rec	**Cam**	**Tape**	RECORDER	CAMERA	TAPE	**…**
u	A	B	C				…
director				D	E	F	…
…							

With :

```
A = RS(this, Cam, Tape)
B = RS(Rec, this, Tape)
C = RS(Rec, Cam, this).
D = RS(this, CAMERA, TAPE)
E = RS(RECORDER, this, TAPE)
F = RS(RECORDER, CAMERA, this).
```

A, B and C are the representations of the right to record a scene on the tape **`Tape`** with the recorder **`Rec`** and with the camera **`Cam`**. So user `u` is authorised to realise this high-level operation, but only with these specific objects.

In the same way, D, E and F are the representations of the right to record a scene on any tape (any instance of class TAPE) with any recorder (any instance of class RECORDER) and with any camera (any instance of class CAMERA). In this example, any user with the role "`director`" can execute the high-level operation with any objects of these classes.

4.2.4. Capability creation rules

When an object is authorised to realise a particular operation (either accessing one method of one object or executing a high-level operation involving several objects), it must be given at least one **capability**. The **capability creation rules** specify how to build such capabilities. These capabilities are similar to those defined in [Tanenbaum *et al.* 1986, Gong 1989]. A capability on an object holds, as a minimum, a reference and a list of rights for that object. We also have to ensure that a capability cannot be forged, cannot be replayed and can only be used by the object for which the capability is created. We assume in the rest of this report that all these properties are implemented by the capability creation and verification algorithms.

Definition of capability creation rules:

Capability creation rules for elementary operations are represented as follows:

$$\mathtt{Cap}(\mathtt{o},\mathtt{o'}.\mathtt{method}(\mathtt{x}_1,\ldots,\mathtt{x}_n)) ::= \{\mathtt{ref}_\mathtt{o},\mathtt{ref}_{\mathtt{o'}},\mathtt{nonce},\mathtt{ref}_{\mathtt{x}_1},\ldots,\mathtt{ref}_{\mathtt{x}_n}\}$$

The term `nonce` means "used only once" and contributes to prevent a capability from being forged[5] and replayed. In this definition, `o` represents the object authorized to access method `method` of `o'`. The arguments of `method` invocation can be constrained to be in the list[6] $\mathtt{x}_1,\ldots,\mathtt{x}_n$.

This capability is a proof that object `o` is authorised to execute `method` of object `o'` with arguments $\mathtt{x}_1,\ldots,\mathtt{x}_n$. This capability will be verified by the security kernel of the site where `o'` is located when object `o` invokes a method of `o'`.

Capability creation rules for high level operations are represented as follows:

$$Cap(o,\mathbf{op(arg_1,\ldots,arg_n)}) ::= \{list\_of\_capabilites,[list\_of\_vouchers]\}$$

We can note that in the definition of a capability, we only consider objects, not users, roles or classes. As a matter of fact, each operation in the system is realised by an object. Either an object or a user may ask for the authorisation to realise a high level operation in the system. In the case of a user, this authorisation may be granted because the user himself or one of his roles has the corresponding rights. In the case of an object, the authorisation may be granted because

5 To prevent forgery, each capability is typically signed by the private key of the Authorisation Server and is ciphered with the public key of the security kernel which will check the capability.

6 This list can be empty, in which case the capability is valid for any parameter, or other constraints can be imposed, e.g., for degradable capabilities (see Section 3.2.2).

the object itself or an ancestor of its class has the corresponding rights. But in each case, the operation is realised by means of an object. So each capability that is created, is created for a specific object. When a user `u` is authorised to realise a high level operation, he starts this operation by means of an interface object (a shell for example). This interface object receives the first capability to start the task.

This capability enables also an object to identify which method of which object is to be invoked to realise a high level operation. For instance, if object `o` is authorised to print a file on printer `p,` the capability will identify the method `printf` of the object `ps` which is the spooler assigned to printer `p` (see Section 4.4).

4.2.5. Vouchers

A capability may contain access rights directly granted to the object receiving the capability, but may also contain indirect access rights called **vouchers**. Vouchers are access rights that are to be granted to an object other than the one receiving the capability. The object that receives such a voucher will transmit it to the object whose identifier is included in the voucher. This second object, when receiving the voucher, will be authorised to perform the operation indicated in the voucher, and will execute this operation with its own identity.

As such, vouchers are analogous to the more usual notion of proxies. A proxy is an access right that an object *O* grants another object *O'* so that *O'* can perform a task in place of *O*. Such proxies are present for examples in Kerberos v5 [Kohl & Neuman 1993], SESAME [Parker 1991] and CORBA [OMG 1995]. Usually, a proxy contains the identities of the granting object and of the granted object, as well as the access rights that are delegated by the granting object to the granted object. By delegating only specific rights, proxies do not present the drawbacks of the SUID bit used by Unix to allow a subject to impersonate another subject in order to execute an operation: in Unix, the impersonating subject receives *all* the impersonated subject's access rights, instead of just those needed to execute the operation. Thus, proxies enable a better implementation of the least privilege principle.

But to create a proxy, the granting object must already possess the access rights it wishes to delegate to the granted object. With vouchers, this is not necessary: the granter does not necessarily possess the rights that are included in the voucher.

As such, vouchers enable still a better implementation of the least privilege principle than the proxies. Moreover, the object receiving a voucher is authorised to perform the corresponding operation with its own identity rather than with the identity of the granting object, as is usually the case with proxies.

In our access control scheme, vouchers can include simple capabilities but may also include authorisations to perform high-level operations. A voucher is valid only for one use by the indicated object, and cannot be forged or replayed.

4.3. Authorisation service

The authorisation service follows different steps according of the kind of operation for which an authorisation is requested:

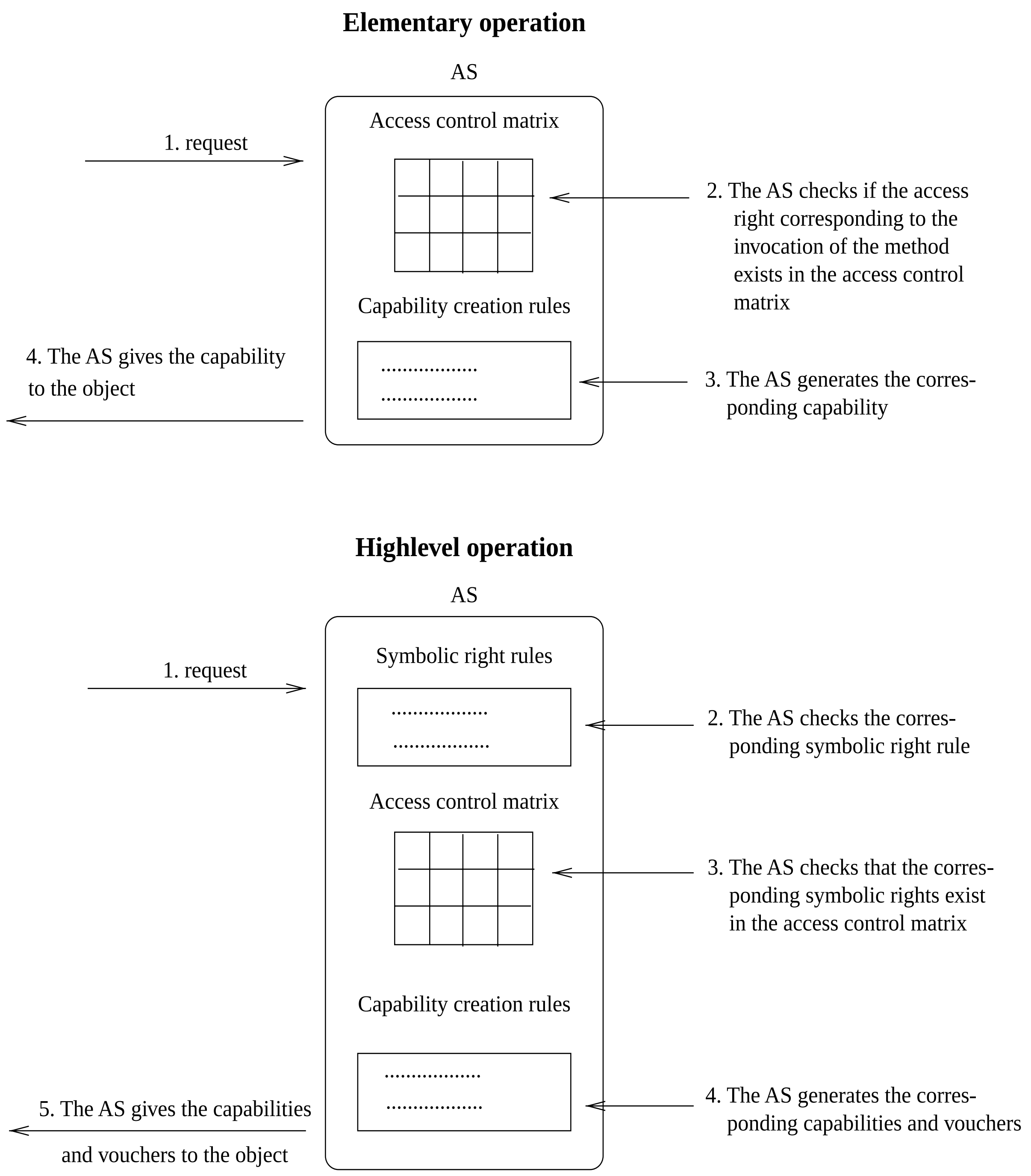
Elementary operation
AS
Access control matrix
1. request
2. The AS checks if the access right corresponding to the invocation of the method exists in the access control matrix
Capability creation rules
4. The AS gives the capability to the object
3. The AS generates the corresponding capability
Highlevel operation
AS
Symbolic right rules
1. request
2. The AS checks the corresponding symbolic right rule
Access control matrix
3. The AS checks that the corresponding symbolic rights exist in the access control matrix
Capability creation rules
4. The AS generates the corresponding capabilities and vouchers
5. The AS gives the capabilities and vouchers to the object

Figure 2: Auhtorisation steps

<u>Steps corresponding to elementary operations:</u>

1 an object wants to invoke a particular method of a particular object;

2 the authorisation server simply checks if the access control matrix holds the right to authorise this operation;

3 if the access is authorised, the authorisation server generates the capability according to the corresponding capability creation rules.

Steps corresponding to high-level operations:

1. a user wants to carry out a high-level operation; this operation is carried out by the execution of an activity;
2. the object representing the user sends a request to the authorisation server to obtain the privileges corresponding to this high-level operation;
3. the authorisation server checks the *symbolic right rules* and the *access control matrix* to see if the user is authorised or not to carry out this operation;
4. if the access is authorised, the authorisation server generates the *capabilities* and *vouchers* according to the corresponding *capability creation rule*; it gives these capabilities and vouchers to the requesting object; the first capability identifies the first object and its first method involved in the execution of the activity;
5. the object does the first method invocation(s) and then transmits the vouchers to other objects involved in the execution of the operation; these other objects may execute their own operations either by using the vouchers or by asking the authorisation servers some other capabilities (and/or vouchers).

Figure 2 summarises these two scenarios.

4.4. Complete example

In the following example, we consider that a user `u` wants to print file $\texttt{f}_3$ on printer $\texttt{p}_4$. The objects which take part in the execution of this action are represented in the following figure. In this figure, `u` is a user of the system, $\texttt{ps}_1$ a print server of class `PRINTSERVER`, $\texttt{fs}_2$ a file server of class `FILESERVER`, $\texttt{f}_3$ a file of class `FILE`, $\texttt{p}_4$ a printer of class `PRINTER` and `tf` a transient file located on the site of $\texttt{ps}_1$.

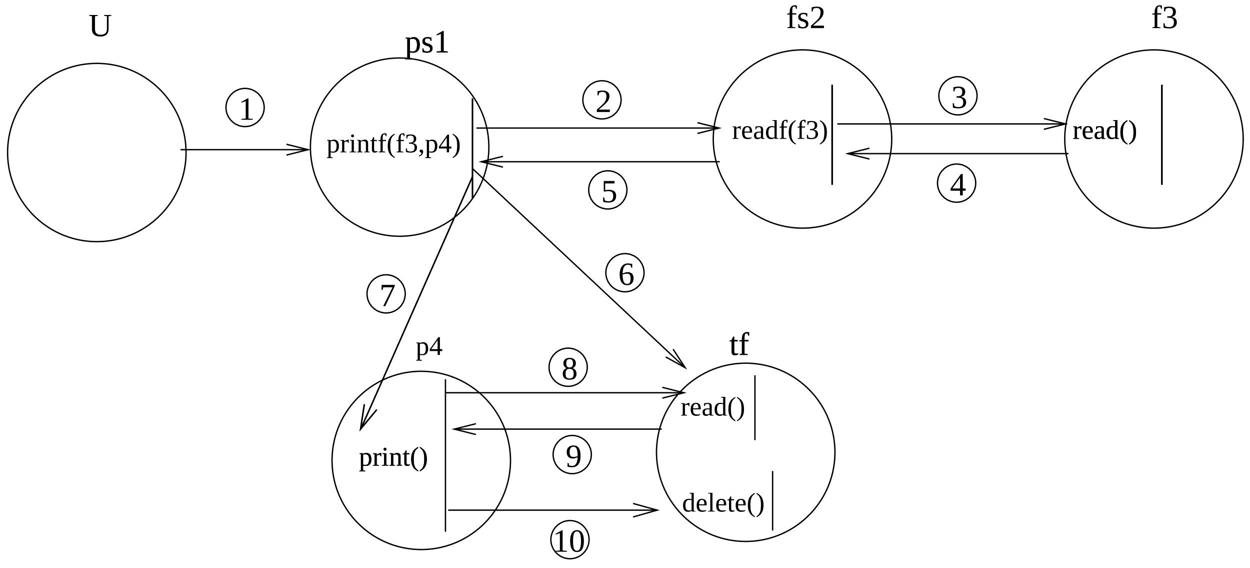

Figure 3: Example of a high-level operation

The different components we presented in the previous section are detailed below: access control matrix, symbolic right rules and capability creation rules.

4.4.1. Access control matrix

	ps1	fs2	fn	f3	p4
U			RF(this)	PF(this,PRINTER)	PF(FILE,this)
ps1					print
fs2			read,write		

Definition of the classes in the symbolic rights declared above:
RF(f): Class(f) ∈ SubClass(FILE)
PF(f,p):(Class(f)∈ SubClass(FILE))∧(Class(p)∈SubClass(PRINTER))

The first row of the access matrix represents symbolic rights that are granted to user u. The symbolic right PF(this,PRINTER) in column f_3 means that u is authorised to print file f_3 on any printer of the system (any instance of the class PRINTER). In the same way, PF(FILE,this) in column p_4 means that u is authorised to print any file of the system on printer p_4. Finally, the symbolic right RF(this) in column f_n means that u is authorised to read file f_n.

The last rows of the access matrix represent method rights that ps_1 and fs_2 have respectively on p_4 and f_n: ps_1 is authorised to invoke method print of the object p_4 and fs_2 is authorised to invoke methods read and write of f_n.

The three lines following the description of the matrix indicate that the symbolic right PF is defined to apply to particular classes: the first parameter must be an instance of class FILE or an instance of a descendant of class FILE. The second parameter must be an instance of the class PRINTER or an instance of a descendant of class PRINTER. In the same way, RF is a symbolic right applying to an instance of class FILE or of a descendant of FILE.

4.4.2. Symbolic right rules

The following rule must be included in the symbolic right rule set:

(SR1) $$\frac{PF_{e,f}(f,p), PF_{e,p}(f,p)}{\mathbf{A(printfile)(e,f,p)}}$$

This rule means that in order to authorise an entity to realise the high-level action "printing file f on printer p" (i.e., the precondition of **printfile(e,f,p)** is true), it must be checked that the entity holds both the following symbolic rights in the access control matrix:

- PF for file f on a set of printers which includes p.
- PF on printer p for a set of files which includes f.

In the rest of the report, we will often designate a high-level action by its representation. We will for instance talk about the high-level action **`printfile(f,p)`** or **`readfile(f)`**.

4.4.3. Capability creation rules

This set of rules includes at least the following rules (in each rule, o represents an object):

(PR1) $\forall$`f` instance of FILE, $\forall$`p` instance of PRINTER, Cap(**`o`**,**`printfile`**`(`**`f`**`,`**`p`**`)`)::= {Cap(o,`PrintServer`(p).`printf(f,p)`), [**`A`**`(`**`readfile`**`)(PrintServer(`**`p`**`),`**`f`**`)`]}

(PR2) $\forall$`f` instance of FILE, Cap(**`o`**,**`readfile`**`(`**`f`**`)`)::= Cap(o,`FileServer(f).readf(f)`)

(PR3) $\forall$`ps` instance of PRINTSERVER, Cap(`o,ps.printf(f,p)`)::= $\{\texttt{ref}_o,\texttt{ref}_{\texttt{ps.printf}},\texttt{nonce},\texttt{ref}_f,\texttt{ref}_p\}$

(PR4) $\forall$`fs` instance of FILESERVER, Cap(`o,fs.readf(f)`)::= $\{\texttt{ref}_o,\texttt{ref}_{\texttt{fs.readf}},\texttt{nonce},\texttt{ref}_f\}$

(PR5) $\forall$`f` instance of FILE, Cap(`o,f.read()`)::= $\{\texttt{ref}_o,\texttt{ref}_{\texttt{f.read}},\texttt{nonce}\}$

(PR6) $\forall$`p` instance of PRINTER, Cap(`o,p.print()`)::= $\{\texttt{ref}_o,\texttt{ref}_{\texttt{p.print}},\texttt{nonce}\}$

Each called method of an object is linked to a capability that is to be given to the object that is authorised to access this method. In this way, $\{\texttt{ref}_o,\texttt{ref}_{\texttt{p4.print}},\texttt{nonce}\}$ is the capability that allows method `print` of printer $\texttt{p}_4$ to be accessed. $\texttt{ref}_e$ represents a reference which allows entity `e` to be ientified.

Each high-level action is also linked to a capability to access a method of an object. As a matter of fact, when an object is authorised to carry out a high-level action in the system, it must start this action by invoking a particular method of a particular object. It is thus given the corresponding capability. The first part of rule (PR1) means that the capability corresponding to the action **`printfile(f,p)`** is the capability to access method `printf` of the print server of printer `p`, while rule (PR2) indicates that the capability corresponding to the action **`readfile(f)`** is the capability to access method `readf` of the file server of file `f`. The notations `PrintServer`(p) and `FileServer`(f) refer to information indicating which printer is the print server of `p` and which file server is the file server of `f` *(this information may be stored thanks to a directory server for example)*.

Furthermore the capability linked to an action may hold another right that has to be delegated to another object of the system. For instance, the voucher [**`A`**`(`**`readfile`**`)(`**`PrintServer`**`(`**`p`**`),`**`f`**`)`] in the capability linked to the action **`printfile(f,p)`** represents the authorisation for the print server of printer `p` to realise the operation **`readfile(f)`**[7]. This voucher is inserted in the capability returned to the object

[7] Let us note that this voucher creation does not require that u holds the symbolic right readfile on f in the access matrix (this is not checked by the symbolic right rule corresponding to printfile). This enforces the least privilege principle, since it is not necessary to be authorised to read a file to be authorised to print it

authorised to carry out the action **`printfile(f,p)`**. This object will pass the voucher to the print server of `p` when it accesses its method `printf`.

In our example, the first request of user `u` is a request to obtain the capability corresponding to the high-level operation **`printfile`**$(\mathbf{f_3},\mathbf{p_4})$. The first row of the access matrix indicates that user `u` is authorised to print file f_3 on any printer of the system and that user `u` is authorised to print any file on printer p_4. Symbolic right rule (SR1) indicates that the high-level operation can be started. A capability is then built according to the rules of the capability creation rules. This capability is constituted of the capability to access method `printf` of ps_1 and of a voucher for ps_1 to perform the high-level operation **`readfile`**$(\mathbf{f_3})$.

4.4.4. Complete scenario

The scenario that enables user `u` to print file f_3 is described here. We indicate in this scenario the roles of the authorisation server (AS) and the security kernels (SK) with respect to capability distribution and verification.

- User `u` wants to print file f_3 on printer p_4 . In the access matrix (cf. Section 4.4.1), `u`'s row holds the symbolic rights $PF(f_3,PRINTER)$ and $PF(FILE,p_4)$. This symbolic right rule (SR1) authorises this access.
- According to the capability creation rule (PR1), the AS gives user `u` the capability $\{Cap(o,ps_1.printf(f_3,p_4)), [\mathbf{A}(\mathbf{readfile})(\mathbf{ps_1},\mathbf{f_3})]\}$ (i.e., the shell `o` started for user `u` receives this capability).
- `u` calls method `printf` of ps_1 (**message 1**) and presents the capability $Cap(o,ps_1.printf(f_3,p_4))$. This invocation is controlled by the security kernel SK, located on the site of ps_1, which checks that the capability is valid. Then `u` gives ps_1 the voucher $[\mathbf{A}(\mathbf{readfile})(\mathbf{ps_1},\mathbf{f_3})]$.
- ps_1 then asks the AS for authorisation to execute the action **`readfile`**$(\mathbf{f_3})$ and presents the voucher received from `u`.
- The AS first checks that the voucher is valid (i.e., is a right for ps_1 created by the AS). Then the AS identifies fs_2 as the file server for f_3 , and finally gives ps_1 the capability corresponding to the action **`readfile`**$(\mathbf{f_3})$, which is $Cap(ps_1,fs_2.readf(f_3))$ (according to the capability creation rule (PR2)). The AS also tells ps_1 the object and the method it can invoke with this capability (i.e., fs_2 and `readf`).
- ps_1 calls method `readf` of fs_2 and presents $Cap(ps_1,fs_2.readf(f_3))$ (**message 2**). The SK located on the site of fs_2 checks the validity of this capability.
- fs_2 asks the AS for a right to access method read of file f_3
- The AS checks that fs_2 is authorised to access method read of f_3 and gives fs_2 the capability $Cap(fs_2,f_3.read())$.
- fs_2 invokes method read of file f_3 (**message 3**). The SK located on the site of f_3 checks the validity of the capability $Cap(fs_2,f_3.read())$.
- ps_1 receives data from file f_3 by the way of two replies (**message 4 and 5**).

- ps_1 creates a temporary file `tf` to be used by p_4 as a spooler. When creating `tf` , ps_1 receives the owner capability on this file from the local SK. Then ps_1 copies f_3 into `tf` by the way of **message 6** (the write access to `tf` is authorised because ps_1 presents the owner capability to the security kernel).
- ps_1 asks the AS for a capability to access method `print` of p_4.
- The AS checks that ps_1 is authorised to access method `print` of p_4 and gives ps_1 the capability `Cap(ps1,p4.print())`.
- ps_1 calls method `print` of p_4 (**message 7**), then asks the security kernel to transmit to p_4 the capabilities to access methods `read` and `delete` of file `tf` (by presenting the owner capability to the security kernel).
- p_4 calls method read of tf (**message 8 and 9**). This invocation is controlled by the local SK, which checks that p_4 holds the capability for method `read` of `tf` . Then p_4 prints the file. After printing, p_4 sends a request to delete `tf` (**message 10**).

5. Extensions

The authorisation scheme presented here is able to implement all the classical security policies, such as Bell-LaPadula [Bell & LaPadula 1976], BiBa [Biba 1977], Clark-Wilson [Clark & Wilson 1987], the Chinese Wall [Brewer & Nash 1989], RBAC [Sandhu *et al.* 2000], etc. Nevertheless, a direct implementation of some of these policies (at least the multilevel policies, such as Bell-LaPadula's and Biba's policies) would require frequent modifications and checks of the access control matrix, and thus frequent requests to the authorisation servers. A more efficient implementation would involve labels attached to each method invocation and each object. An example of such a multilevel implementation of a similar authorisation scheme is proposed in [Nicomette & Deswarte 1996].

Let us note that the degradable capability concept introduced in our authorisation scheme contributes to reducing the frequency of changes and checks of the access control matrix. But this kind of capability is more complex than other capabilities since they impose constraints on invocation parameter values (see footnote on page 19). On the other hand, this kind of constraints offers more flexibility in the authorisation scheme. For instance, threshold or ceiling values can be introduced in the privileges recorded in the access control matrix, to represent the privilege to grant a loan up to a certain amount.

6. Relations with the other workpackages

The implementation of the authorisation services will make use of the transaction primitives, dependable communication protocols and threshold cryptography developed in WP2 (Dependable Middleware). It will also use the implementation of intrusion-tolerant Certification Authorities in WP4 (Dependable Trusted Third Party) to realise a dependable authentication server.

7. References

[Ahad *et al.* 1992]
R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marinowski and E. Onuegbe, "Supporting Access Control in an Object-Oriented Database Language", in *3rd International Conference on Extending Database Technology (EDBT)*, (Vienna, Austria), LNCS, pp.184-220, Springer-Verlag, 1992.

[Bell & LaPadula 1976]
D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, The MITRE Corporation, Tech. Report, N°MTR-2997 (ESD-TR-75-306), March 1976.

[Bertino 1992]
E. Bertino, "Data Hiding and Security in an Object-Oriented Database System", in *8th IEEE International Conference on Data Engineering,* (Phoenix, Arizona, USA), pp.338-47, 1992.

[Biba 1977]
K. J. Biba, *Integrity Consideration for Secure Computer Systems*, The MITRE Corporation, Technical Report, N° MTR-3153, Rev. 1, June 1977.

[Biskup & Brüggemann 1988]
J. Biskup and H. H. Brüggemann, "The Personal Model of Data", *Computers & Security*, 7 (6), pp.575-97, 1988.

[Blain & Deswarte 1990]
L. Blain and Y. Deswarte, "Intrusion-Tolerant Security Server for Delta-4", in *ESPRIT 90 Conference*, (CEC-DG-XIII, Ed.), (Brussels (Belgium)), pp.355-70, Kluwer Academic Publishers, 1990.

[Brewer & Nash 1989]
D. F. C. Brewer and M. J. Nash, "The Chinese Wall Security Policy", in *Proc. Int. Symp. on Security and Privacy,* (Oakland, CA, USA), pp.206-14, IEEE Computer Society Press, 1989.

[Clark & Wilson 1987]
D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", in *Proc. Int. Symp. on Security and Privacy,* (Oakland, CA, USA), pp.184-94, IEEE Computer Society Press, 1987.

[Deswarte *et al.* 1991]
Y. Deswarte, L. Blain and J.-C. Fabre, "Intrusion Tolerance in Distributed Computing Systems", in *Proc. Int. Symp. on Security and Privacy*, (Oakland, CA, USA), pp.110-21, IEEE Computer Society Press, 1991.

[DoD 1985]
Trusted Computer System Evaluation Criteria (TCSEC), Department of Defense, USA, Report N°DoD 5200.28-STD, 1985.

[Erlingsson & Schneider 2000]
U. Erlingsson and F. B. Schneider, "IRM Enforcement of Java Stack Inspection", in *IEEE Symposium on Security and Privacy,* (Berkeley, CA, USA), pp.246-55, 2000.

[Gasser *et al.* 1989]
M. Gasser, A. Goldstein, C. Kaufman and B. Lampson, "The Digital Distributed System Security Architecture", in *12th National Computer Security Conference,* pp.305-19, NCSC and NIST, 1989.

[Gasser & McDermott 1990]
M. Gasser and E. McDermott, "An Architecture for Practical Delegation in a Distributed System", in *IEEE Symposium on Security and Privacy,* (Oakland, CA), pp.20-30, 1990.

[Gong 1989]
L. Gong, "A Secure Identity-Based Capablity System", in *IEEE Symposium on Security and Privacy,* (Oakland, CA, USA), pp.56-63, 1989.

[HP 1998]
HP, *HP Praesidium Authorization Server 3.1: Increasing Security Requirements in the Extended Enterprise*, November 2 1998
(http://www.hp.com/security/products/authorization_server/papers/whitepaper/).

[Kohl & Neuman 1993]
J. Kohl and C. Neuman, *The Kerberos Network Authentication Service (V5)*, Internet RFC, N°1510, 1993.

[NCSC 1987]
Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria, National Computer Security Center, Report N°NCSC-TG-005, 1987.

[Nicomette 1996]
V. Nicomette, *La protection dans les systèmes à objets répartis,* Thèse de doctorat, Institut National Polytechnique de Toulouse, Décembre 1996 (Aussi Rapport LAAS n°96496).

[Nicomette & Deswarte 1996]
V. Nicomette and Y. Deswarte, "A Multilevel Security Model for Distributed Object Systems", in *Proc. 4th European Symposium on Research in Computer Security (ESORICS 96),* (Rome, Italy), LNCS n°1146, pp.80-98, Springer-Verlag, 1996.

[Nicomette & Deswarte 1997]
V. Nicomette and Y. Deswarte, "An Authorization Scheme for Distributed Object Systems", in *Proc. Int. Symposium on Security and Privacy,* (Oakland, CA, USA), pp.21-30, IEEE Computer Society Press, 1997.

[OMG 1995]
OMG, *CORBA Security*, OMG TC Document, N°95-12-1, December 1995.

[Parker 1991]
T. Parker, “A Secure European System for Applications in a Multi-vendor Environment (The SESAME project)”, in *14th National Computer Security Conference,* (Washington (DC, USA)), pp.505-13, NCSC and NIST, 1991.

[Reiter *et al.* 1994]
M. K. Reiter, K. P. Birman and R. V. Reness, “A Security Architecture for Fault-Tolerant Systems”, *ACM Transactions on Computer Systems,* 12 (4), pp.340-71, November 1994.

[Richardson *et al.* 1992]
J. Richardson, P. Schwarz and L.-F. Cabrera, “CACL: Efficient Fined-Grained Protection for Objects”, in *ACM Object-Oriented Programming Systems Languages and Applications (OOPSLA),* (B. Thuraisingham, R. Sandhu and T. C. Ting, Eds.), (Vancouver, Canada), pp.263--75, Springer-Verlag, 1992.

[Rushby & Randell 1983]
J. Rushby and B. Randell, “A distributed secure system”, *IEEE Computer,* 16 (7), pp.55-67, July 1983.

[Sandhu *et al.* 2000]
R. Sandhu, D. Ferraiolo and R. Kuhn, “The NIST Model for Role-Based Access Control: Towards a Unified Standard”, in *5th ACM Workshop on Role-Based Access Control,* (Berlin,Germany), pp.47-63, 2000.

[Tanenbaum *et al.* 1986]
A. S. Tanenbaum, S. J. Mullender and R. v. Renesse, “Using Sparse Capabilities in a Distributed Operating System”, in *6th International Conference on Distributed Computing Systems,* (Cambridge, MA, USA), pp.558-63, 1986.

[Welch & Stroud 2000]
I. Welch and R. Stroud, “Using Reflection as a Mechanism for Enforcing Security Policices in Mobile Code”, in *6th Euroopean Symposium on Research in Computer Security (ESORICS 2000),* (F. Cuppens, Y. Deswarte, D. Gollmann and M. Waidner, Eds.), (Toulouse, France), LNCS 1895, pp.309-23, Springer Verlag, 2000.

[Wulf *et al.* 1974]
W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, “HYDRA: The Kernel of a Multiprocessor Operating System”, *Communications of the ACM,* 17 (6), pp.337-45, 1974.

[Zurko *et al.* 1999]
M.-E. Zurko, R. Simon and T. Sanfilipo, “A User-Centered, Modular Authorization Service Built on an RBAC Foundation”, in *IEEE Symposium on Security and Privacy,* (Berkeley (CA, USA)), pp.57-71, 1999.