DIVISION OF COMPUTER SCIENCE

Roles and Rights

Marie Rose Low
Bruce Christianson

Technical Report No.232

October 1995

ROLES & RIGHTS

Marie Rose Low and Bruce Christianson

M.R.Low@herts.ac.uk B.Christianson@herts.ac.uk

Abstract

In this paper we describe and discuss a protection model designed by Coulouris and Dollimore to protect objects which are shared by different principals participating in a co-operative task. We then show that the Self-Authenticating Proxy mechanism would enable a satisfactory implementation of this model as well as providing other benefits which had not previously been envisaged.

1. Introduction

In this paper we discuss a protection model designed by Coulouris and Dollimore [1] and then propose a way to implement this model. The protection model 'is intended for use in an open environment in which co-operative tasks are carried out using co-operative applications accessing shared objects in a distributed environment'. The model addresses the need to protect objects used within tasks that are to be carried out by more than one principal. In a co-operative task, where principals are not necessarily equally trusted, it is important to be able to specify different access rights to objects for different principals. The model assumes that applications are not responsible for the protection of the objects they manipulate, that shared objects provide interfaces through which their operations may be executed, and that the objects protect themselves according to a pre-defined set of access rights.

This model is based on the following principles. Firstly that the operations that each principal needs to perform in a task can be assigned to a symbolic role name i.e. each principal plays a particular role in a task. Secondly, that security policies can be specified in terms of the operations that need to be performed on objects by individuals acting in particular roles. Therefore, users accessing shared, protected objects, act within a task structure and the access rights to those objects are defined according to the needs of each role within the task. Finally, the protection model has two levels. At the specification level, security requirements are specified in terms of generic operations which are independent of the applications used. At the programming level, access control is applied to object operations at the object server interface in the execution environment.

The aim of this paper is to show that an existing protection scheme, Self-Authenticating Proxies (SAProxies) [2], may be used to implement this model successfully. The following section describes the protection model in greater detail. The subsequent sections provide a brief description of the SAProxy scheme, an illustration of how the scheme can be applied and a discussion of the benefits of using such a scheme.

2. The Protection Model

Coulouris and Dollimore propose that the security requirement of any task which operates on objects should be specified by users in terms of the generic operations that each role is allowed to perform on the objects in order to complete that task. The right

to perform generic operations, access rights, are granted not to particular individuals but to the symbolic role names.

In [1] the security requirements for a task are defined in a security template. The template expresses the security policy in terms of which roles are allowed to perform what operations. Objects are grouped according to the role that created them so as to reduce the complexity of the security template and also because a view is adopted that the rights of roles to access objects may be related to the roles that created those objects.

Users specify access rights to each group of objects created by a role in terms of the generic operations e.g. read, write, that other roles are allowed to perform on the group of objects. When an object is created the access rights for that object are taken from the security template. These rights are attached to the object, effectively as an ACL.

Each role is granted authority to access those objects that it must operate on to play its part in the execution of that task. When the task is to be executed, a principal is bound to each role and each principal takes on the rights granted to that role for the duration of the task. A principal's authority to act in a role must be verifiable either by a trusted task manager or by checking client supplied credentials. Only when a principal is bound to a role can the authority granted to the role be exercised.

Using roles to define the set of operations to be performed, and thus the set of access rights required, by principals in a task, enables the security requirements for a task to be defined before knowing which principals are to play those roles. Furthermore, the definition of a role may be modified prior to those rights being exercised without having to modify an access control list entry for each principal who might act in each role. The same security template may also be used for more than one instance of the same task but with different users bound to each of the roles.

As well as using the security template to apply 'generic access control' to operations, the two level model also defines programming level access control. Each class of object has its own programmable operations. These operations are managed by an object server and are defined in the object server interface. Roles are defined with generic rights, not in terms of these programmable object operations. Each programming level operation specified, however, is mapped onto one or more generic operations. This mapping is implemented at the object server interface. When an object operation is requested, these generic operations are checked against the generic rights that the role has been granted as specified in the object's ACL. Only those object operations which can be mapped completely onto the generic rights granted to a role can be executed by that role. For example, a role that has only been granted the generic right to read an object, may not be used to execute an edit operation on the object as this would also require the write right.

The user that specifies the rights of a role does not have to be aware of the full diversity of object operations that may be available. Thus, the process of rights allocation is simplified, but at the cost of specifying the mapping from the programming level object operations to the generic operations.

The way the protection model can be used is illustrated by the following example.

2.1 An Object Server

Coulouris and Dollimore illustrate the two-level protection model by describing how it could be used by a server for persistent objects. In this example, there is a task service which manages the protection needs of the object server. This service maintains a security template for each type of task to be carried out and also checks the authority of principals to act in a role. The access rights to objects are specified by the users in the security template to suit the requirements of the task.

When a new object is created, the object server obtains the access rights for the object from the security template maintained by the task service. This information is then attached to the new object. Further changes to the security template, therefore, do not affect objects already created. The object operations are defined in the server interface

and managed by the object server. Each definition includes the mapping of the object operation onto the generic operations. When a request arrives the object server checks with the task service that the user presenting the request is allowed to play the role in that task. The object server then checks that the role making the request has been granted the generic rights that correspond to the mapping from the object operation to the generic operations. If the principal's authority to act in the role is valid and the role has been granted the generic rights required, then the request can go ahead.

In order to implement this protection model, a mechanism is required which is able to define roles in a task and their rights; to verifiably bind a principal to a role; to specify the security requirements of a task in terms of those roles and their rights; to verify the mapping from programming to generic operations; to apply further access control at the object operation level.

In the following sections, we discuss the notion of delegation of authority and show how the SAProxy scheme [2, 3], based on tokens of delegation, can be used to implement the model.

3. Roles and Delegation

Consider now the argument that a symbolic role name can be viewed as a delegatee and that when access rights are granted to a role, it is the equivalent of delegating authority to that role. A principal, e.g. the owner of an object, may have authority over the object such that the principal is empowered to perform operations on the object e.g. he has read/ write access to the object. This principal is then in a position to grant all or some of these access rights to other principals. Granting rights to other principals can be seen as delegation of authority. As has been seen in section 2, specifying access rights is the same as granting authority to perform operations i.e. delegation of authority

A role without the right to perform any operation is not a useful role. Giving access rights to a symbolic role name, the equivalent of delegating authority to it, makes a role useable. Thus, the creation of a useful role demands the delegation of authority to that role. Therefore a role can be seen as a delegatee to whom rights have been granted.

The advantages of using symbolic role names are that tasks can be clearly specified, operations defined and authority delegated without having to decide which particular principals are to participate in the task until the task is to be executed. The difference between delegating to a principal and delegating to a role arises only when the delegated authority is to be used. A role is 'merely' a collection of 'permitted operations', a set of access rights. This set of rights, the symbolic role name, has not got the ability to exercise this authority by itself and so requires that a principal is bound to that symbolic name before any operations can be performed. Binding a role to a principal is also delegation - the rights of the role are delegated to the principal. The obvious outcome of considering a role as a delegatee is that, as there need be no distinction in dealing with a role name or a principal until an operation is requested, a delegation scheme may be used to define and enforce the protection requirements of a task even when this is specified in terms of roles.

The protection model discussed above can be viewed a series of delegations of authority which permit certain principals to participate in co-operative tasks. It would therefore appear plausible that a delegation mechanism is a suitable basis for an implementation of this model. The SAProxy scheme provides a delegation mechanism which enables principals in authority to delegate that authority in a manner which is enforceable, where the use of this authority is verifiable and where the delegator is assured that his authority will not be abused.

3.1 The Model in terms of Delegation

The model proposed in [1] defines the functions that need to be performed but does not specify how they are to be carried out e.g. how does a user actually make an entry in a

security template, who authorises a user to act in a role, how is this expressed and how it is verified.

Consider now the functions performed by users, the object server and task service described in 2.1 and how these functions may be viewed in terms of the delegation of authority and so represented using a delegation scheme. In order to enable free distribution of responsibilities over a distributed environment the different functions that are carried out by the principals and the task service may, where possible, be allocated to distinct parties e.g. different parties may give a role its generic rights and others may bind a role to a principal. However, there is no reason why some or all these functions should not be performed by the one party if this satisfies the needs of a particular application. As will be shown these functions can be performed by principals with the relevant authority, i.e. users who have rights to an object, and the object server; there is no need for a trusted task service.

The number of distinct parties and the authority each has is a matter of co-operative policy between those responsible for the task to be carried out. In the example under consideration, there must exist at least one principal G, authorised to grant generic rights to a role R in a particular task, a principal L, authorised to bind roles to principals, and a principal P, who defines the mapping from object operations to generic operations. Principals G, L and P may or may not be the same. Then there must be principals that execute the task e.g. principal Z takes on role R to execute task T.

The principal, G, delegates its authority over the objects and grants generic access rights to the roles e.g. R, in the task. When binding a role R to principal Z, L provides Z with the credentials to act in the role R i.e. L delegates to Z the authority to act in role R. The principal P specifies which generic operations need to be invoked by each programmable object operation[1]. Z's authority to act in the role R, the rights granted to R by G and the mapping from object to generic operations may all be represented as electronic tokens of delegation and these can be made available to the object server with the request made by Z. The object server can then verify that these tokens are valid, that Z's request is within the authorised constraints and that the generic rights granted to R are within those specified in the mapping from the programming level rights (i.e. the object operation requested by Z) to the generic rights.

In [1], authority is delegated to principals only, roles are defined with rights to perform certain functions and programmable operations are specified to use particular generic operations. However, we have shown above that delegation can be applied not only to principals, but also to roles and programmable operations although these need to be bound to or invoked by a principal before the authority delegated can be exercised. By adopting this view the following is possible. More than one party can give rights to a role; these rights can be dynamically assigned to the role; the rights required to perform an object operation need not be hard coded into the object server interface and finally, a consistent approach is used at all levels of authorisation which makes the enforcement and auditability of this authorisation scheme easy to implement and verify.

With the SAProxy scheme, this authority and binding can be expressed in terms of SAProxy tokens.

4. The SAProxy Scheme

The SAProxy scheme is a mechanism, based on tokens of delegation and public key encryption [4]. This scheme allows principals to define and delegate access rights for objects over which they have authority, to define roles and to grant a principal authority to act in a role.

---

[1] In terms of delegation, P grants each object operation the right to use the generic operations which are necessary to the object operation.

An SAProxy is an extended capability. Like a capability it states the access rights allowed by the token. An SAProxy token, however, holds more information than a pure capability and includes an identifier for the issuer of the capability, an identifier for the principal to whom the capability was issued and a life span[2]. Each SAProxy is digitally signed by its issuer. In this way each token i.e. delegation of authority, is unforgeable and attributable, because a party's secret key is never shared. SAProxies are theft-proof. Only the owner of a SAProxy, i.e. the party to whom the token was issued and is named in it, can use it [2,6].

SAProxies may be bound together in order to express further delegation of authority and combined authorisation. Thus, a SAProxy field may refer to another SAProxy instead of holding a real value. SAProxies are referred to by their digital signatures which provide a unique and verifiable pointer mechanism. This enables a service to keep a secure cache of frequently used SAProxies, thus improving performance by reducing the number of SAProxies that need to be transmitted and verified for a request.

All SAProxies necessary to show the authority of the requester are presented with the request for an operation, together with the public keys needed to verify the tokens. A SAProxy is self-authenticating because any principal with knowledge of the token issuer's public key can verify it. The authority, expressed in unforgeable SAProxy tokens, can be produced and verified locally by each principal and can be recorded to provide an audit trail of all requests and the authority under which each action is performed.

4.1 The SAProxy Mechanism

SAProxies are based on public key encryption, so all principals involved must have a public key pair. Each public key has to be certified by an authority which the owner of the key trusts and in which other parties in the scheme have sufficient trust for the task they are trying to protect [5]. The certification authority, CA, has its own public key pair, public key K_{CA}^+ and private key K_{CA}^-, and produces a public key certificate (PKC) for a principal's public key. A principal's public key and the owner's identity are bound together in a token and digitally signed with the private key, K_{CA}^-, of the certification authority. The resulting signature block forms part of the PKC.

A PKC, C_A, for a principal A with public key K_A^+ and private key K_A^- is of the form:

C_A: A K_A^+ CA K_{CA}^+ life-span $sigC_A$

where the PKC signature block is

$sigC_A$: {A K_A^+ CA K_{CA}^+ life-span}K_{CA}^-

A principal can then be identified and referred to by his PKC, because these are unique, and the PKC itself can be referred to by its signature block, $sigC_A$ [6].

The tokens, the SAProxies, represent the access rights to an object as delegated by the issuer of the SAProxy, to a party named in the SAProxy. Each SAProxy is also signed by the issuer and the signature block also forms part of the token.

To briefly illustrate the use of SAProxies, consider a principal A, with PKC C_A, which has authority over object O and wishes to grant write access to principal B with PKC C_B. The SAProxy, $D_{B:A}$, would be as follows:

$D_{B:A}$: $sigC_B$ $sigC_A$ O *write* $sigD_{B:A}$

where the SAProxy signature block is

$sigD_{B:A}$: {$sigC_B$ $sigC_A$ O *write*}K_A^-

---

[2] For the sake of clarity, the life span of SAProxies will not be shown in their format in the following section.

Anyone with A's PKC can use A's public key to verify that A signed the SAProxy and that it has not been tampered with.

Within a single SAProxy, the information shown above need not be explicitly stated but may be referred to by other SAProxies. For example, if B then delegates his own authority to C, B does not refer to O explicitly, but refers to $D_{B:A}$. In doing so, B is automatically binding into his delegation of authority, proof of his own authority. This is shown in B's token, $D_{C:B}$, generated for C:

$D_{C:B}$: $sigC_C$ $sigC_B$ $sigD_{B:A}$ *write* $sigD_{C:B}$

Thus these tokens of delegation can be bound together to show the grounds upon which access to an object is granted. Instead of nesting the whole PKC or SAProxy within a SAProxy, each token is referred to by its signature block [6]. It is important to note that because these SAProxies are signed by each issuer, they can be verified by any party, possessing an SAProxy checker, and not just by the object server.

5. An Implementation

In this section we show how the two level protection model can be implemented using SAProxies.

5.1 Using SAProxies

Each principal involved must first generate a public key pair and then get the public key certificate, PKC, issued by a certification authority they trust (the CA does not generate public key pairs for principals). Principals are then referred to by their PKCs e.g. G's PKC is C_G[3]. G delegates authority to the role R by granting the appropriate generic access rights to R in a delegation SAProxy, $D_{R:G}$.

$D_{R:G}$: R $sigC_G$ object *generic rights* $sigD_{R:G}$

L gives Z the authority to act as R in a visa SAProxy [3], $V_{Z:L}$; this is Z's credentials to take on the role R.

$V_{Z:L}$: $sigC_Z$ $sigC_L$ *null* R $sigV_{Z:L}$

In the above token, L is granting Z, the right to use his PKC, C_Z, in the role, R. The object that role R may operate on is left null, to be dictated by the authorisation granted to R. The principal P specifies the mapping from the object operation to the generic operations. For each programmable object operation, O, P generates a SAProxy, $M_{O:P}$, which gives O the right to invoke particular generic operations. This defines the mapping from programming level rights to generic rights. P, however, has not got authority over objects (G gives access to objects) and so the object field is null. Thus the object operation, O, is given rights to invoke generic operations on whatever objects the calling principal has access to.

$M_{O:P}$: O $sigC_P$ *null* *generic rights* $sigM_{O:P}$

To operate on an object in role R, Z then sends a request SAProxy, $R_{S:Z}$, to the object server, S, for the operation he wishes to perform. In $R_{S:Z}$, Z binds together his right to act as R, as shown by his visa, $V_{Z:L}$, the generic rights granted to R by G, as stated in $D_{R:G}$ and the mapping from programming level rights to generic rights for the required operation O, as defined in, $M_{O:P}$. The server is referred to by its PKC, C_S, in $R_{S:Z}$.

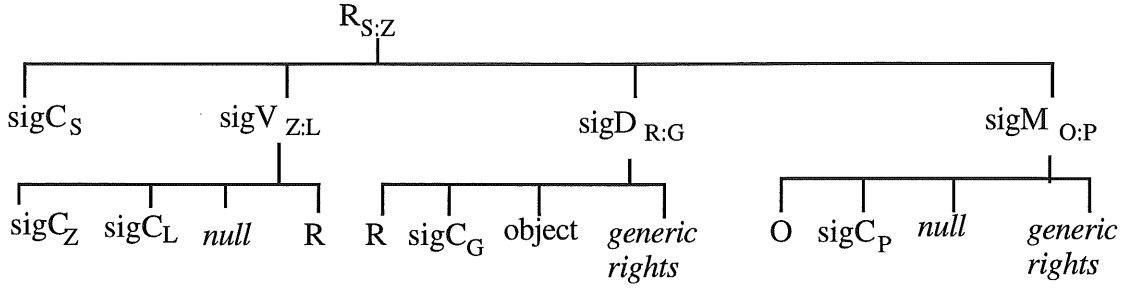
$R_{S:Z}$: $sigC_S$ $sigV_{Z:L}$ $sigD_{R:G}$ $sigM_{O:P}$ $sigR_{S:Z}$

The object server (or a SAProxy checker trusted by the object server) checks that the generic rights defined in $M_{O:P}$ map onto one of the generic rights granted to R in $D_{R:G}$.

---

[3] It is worth noting at this point, that in terms of SAProxies, a role is a principal without a public key pair. Thus a role can be a delegatee, but cannot delegate of it own accord A principal acting in a role R, may, however, delegate the rights granted to R to another principal or role if this is permitted by the definition of the role.

The object server also checks that the role, R, in $D_{R:G}$ is the same role, R, granted to Z in $V_{Z:L}$. The line of authority can be seen in the following tree:



The line of authority through these tokens of authorisation can be checked as can the rights of each party involved. This authority is publicly verifiable and unforgeable because each SAProxy is signed with the secret key of the authorising party.

If the generic rights granted to R change, then Z only has to present the new $D_{R:G}$ with his request to exercise the new rights; his visa SAProxy does not need to be re-issued. Similarly, if the principal Z is replaced by a different principal, Y, the SAProxy, $D_{R:G}$, defining the rights of role R, does not have to be re-issued. The only requirement is that Y is issued with a visa SAProxy granting him the authority to act in role R. Also, the mapping from object operations to generic operations does not have to be hard-coded into the object interface but, instead, is presented at the time the operation is requested.

5.2 Finer Grained Control

In the two level protection model proposed in [1], the operations a role holder may perform at the programming level are determined by the generic access rights granted to that role. As previously mentioned this approach solves the problem of operation diversity, where the detailed nature of the object operations supported may not be generally known to users specifying a role's access rights. However, this approach does not cover all problems. Consider the situation where two operations on objects map onto the same generic right e.g. Edit and Append will map onto the generic right, write. The role R must be granted the generic right, write, to be able to perform an Append operation on an object. As the model stands, it is then not possible to prevent R from performing an Edit operation. Not only does this model have the extra cost of the specification of the mapping from programming level rights to generic rights, but it only enables coarse grained access control of operations on objects i.e. access control is at the level of the UNIX model of file protection, because all roles that have write access can perform all object operations that map onto this right. Fine grained access control, is possible in a UNIX environment by adding an access control list check at the object operation level [7].

A finer granularity of access control can also be defined when the SAProxy scheme is used to implement this model. A principal, X, can have authority over the object operations that a role, R, may perform on an object. X then delegates programming level access rights for each object operation, in a SAProxy, to the role, R. This SAProxy can then be bound together with a principal's authority to act in a role and that role's generic rights. These rights are, as above, granted to R, and not the principal who is to act in that role. Obviously, any programming level access rights granted to R must be within the limits of the generic rights already granted to R by G. A SAProxy from X grants R the authority to perform the required object operation, Append, whilst not allowing him to perform an Edit operation. In this way the right to request object operations is restricted to only those operations that are necessary to complete the task.

X issues R with a SAProxy, $D_{R:X}$, with the right to invoke an object operation e.g. *Append* . The object operation is expressed in terms of the SAProxy, $M_{O:P}$, which maps it on to the appropriate generic operations.

$M_{O:P}$: *Append* $sigC_P$ *null* *generic rights* $sigM_{O:P}$

The SAProxy from X is then as follows:

$D_{R:X}$: R $sigC_X$ object/*null* $sigM_{O:P}$ $sigD_{R:X}$:

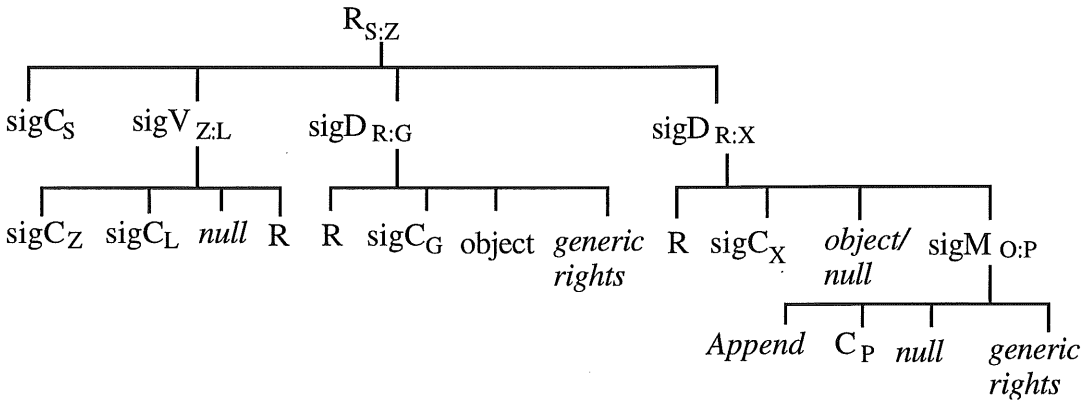
The object field in $D_{R:X}$ may refer to a particular object or maybe left null, depending on the authority X can exercise e.g. X's function may be just to restrict the object operations a role can perform and not necessarily to control the objects on which these operations may be performed.

$D'_{R:X}$ may also refer directly to $D_{R:G}$ and in this way constrain the access rights to a particular object.

$D'_{R:X}$: R $sigC_X$ $sigD_{R:G}$ $sigM_{O:P}$ $sigD'_{R:X}$:

However, if the SAProxies are used as illustrated below, it is then possible to change the objects and rights a role may access, as expressed in $D_{R:G}$, without having to re-issue the higher level programming rights specified by X in $D_{R:X}$. The request, $R_{S:Z}$, to the server is now modified as shown below. Z binds together his right to act as R, as shown by his visa, $V_{Z:L}$, and the programming rights granted to R by X, as defined in $D_{R:X}$ and allowed by $D_{R:G}$.

$R_{S:Z}$: $sigC_S$ $sigV_{Z:L}$ $sigD_{R:G}$ $sigD_{R:X}$ $sigR_{S:Z}$



As the task develops and users that specify access rights become familiar with the object operations, the rights of a role to perform particular operations can be restricted to only those object operations required for the task in hand and not to all operations that map onto the generic rights.

5.3 Role Identification

It is possible that two roles, created for different purposes by different principals, may have the same name. This would allow a principal with a visa SAProxy to act in one of these roles to use the SAProxies giving rights to the other role with the same name. In order to eliminate this possibility, the role name can be bound with the public key of the role creator, Q[4]. (A public key pair is always assumed to be unique.)

A role identifier, R, generated by Q, may then be composed of the following elements:

R: $\{K_Q^+$, role name$\}K_Q^-$

---

[4] Q, the role creator, and G, authorised to grants rights to a role, may in fact the same party. We make the distinction in order to illustrate that there may be more than one G that may grant rights to a role.

Any party with Q's PKC can verify that R has been generated by Q by checking R with Q's public key. No other party can generate the same R because K_Q^- is kept secret. The same principle can be applied to object names so that these can also be uniquely identified. The public key of the object creator/ owner can be bound into the object name.

6. Discussion

We have shown that the SAProxy scheme is able to satisfy the needs of the protection model proposed in [1]. SAProxies can be used to grant rights to a role, and to authorise a principal to act in a role. A principal thus authorised to act in a role can exercise the rights granted to that role. Conversely, a user's actions within a task are restricted to those granted to that role (no rights granted to other roles may be exercised).

SAProxies allow users to define and enforce their own protection policies. The different types of SAProxies and the way they can be chained together may be used as a 'language' to express a security policy. Thus, the security policy for a task, the security template, can be expressed in terms of SAProxies.

The two level access rights can be implemented using SAProxies. Expressing the mapping from programming level rights to generic rights with SAProxies allows greater flexibility than hard-coding them in the object interface. With such an implementation, it is also possible to achieve a finer granularity of access control because access rights can be specified for object operations.

As SAProxies can be easily generated it is not important that a principal know the full diversity of object operations in order to specify programming level rights. The late binding of a principal's credentials (the visa SAProxy) and authority (the delegation SAProxy) at the time an operation is requested, allows generic and programming level access rights to be dynamically assigned to roles. New SAProxies can be issued to accommodate all operations. Specifying rights at the programming level is also more secure than basing access on generic rights because, as it stands, the two level protection model can allow a role to perform object operations that were not intended for that role.

All requests must be accompanied by the authorising SAProxies and these can be verified locally by anyone with the necessary PKCs. If the object server keeps a record of all requests and their SAProxies and PKCs, then this becomes an audit trail which can be used for arbitration and in the management of resources to charge for services. As each token, including requests, is signed with the secret key of the issuer of the token, the server cannot forge a request and a client cannot deny its own actions.

As all SAProxies are verifiable by any party, the responsibility for making security decisions need not lie with the object server. The task of checking the validity of a chain of SAProxies can be assigned to a separate service. This may prove valuable in a wider context, where a server may be accessed from separate management domains and each domain has its own principals who issue SAProxies. Enforcing the security constraints imposed by these SAProxies may be done within each domain by having a local SAProxy checker which can front access to the server. In this way an autonomous domain need not rely on the trustworthiness of a remote object server to enforce its protection policy.

SAProxies are tokens of delegation, so further delegation of authority to other principals can be done naturally. This delegation can itself be restricted by including (or excluding) the right to delegate further in the access rights in the SAProxy. A further requirement envisaged in [1] is that of finalising an object, whereby roles which are allowed to update an object attach a digital signature to the object, thus preventing further modification. This notion is similar to those discussed in [8]. Finalising an object is easily achieved in this implementation. As the SAProxy scheme is based on public key encryption, each principal has a private key which can be used to produce a digital signature for an object which can be checked with the principal's public key in

his PKC, and so prevent further modification of that object. The authority of a principal to finalise an object can be shown in his visa and SAProxies.

We have shown not only that it is possible to implement the two level protection model using SAProxies but also that there are benefits gained in doing so which may not be available in other implementations.

References

[1] Coulouris G., Dollimore J. Protection of Shared Objects for Cooperative Work. *Pre-publication draft,* xx(x):1-16, March 1995.

[2] Low M.R., Christianson B. Self Authenticating Proxies. *Computer Journal,* 37(5):422-428, October 1994.

[3] Low M.R. *Self Defence in Open Systems: Protecting and Sharing Resources in a Distributed Open Environment* Hatfield: University of Hertfordshire, Computer Science Division. Thesis (PhD), September 1994.

[4] Diffie W., Hellman M.E. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644-654, November 1976.

[5] Low M.R., Christianson B. *Authentication v Certification* Hatfield: University of Hertfordshire, Computer Science Division, Technical Report 216, January 1995.

[6] Christianson B., Low M.R. Key-spoofing attacks on nested signature blocks. *IEE Electronics Letters*, 31(13):1043-1044, June 1995.

[7] Low M.R., Christianson B. Fine Grained Object Protection in UNIX *Communications of the ACM Operating Systems Review,* 27(1):33-50, January 1993.

[8] Christianson B., Snook J. *Shrink Wrapped Optimism: The DODA Approach to Distributed Document Processing.* Technical Report 187, Hatfield: University of Hertfordshire, Computer Science Division, 1994.