DIVISION OF COMPUTER SCIENCE

Self Defence in Open Systems using Self Authenticating Proxies

Marie Rose Low

Technical Report No.161

July 1993

Self Defence in Open Systems using Self Authenticating Proxies

Marie Rose Low
School of Information Sciences, University of Hertfordshire.

July 1993

Abstract

Most of the methods currently used to protect both users and systems and their objects and servers, involve authentication of principals and access control of these objects and servers. Authentication and access control are usually implemented as two separate mechanisms because they are logically separate functions. Yet there is little benefit in having one without the other. A consistent approach to both of these functions is proposed in this paper. In this new approach, resource management, (another aspect of protection) can be included. By combining the properties of public key encryption with cascading proxies a single mechanism is devised to provide these three aspects of protection.

1. Introduction

As the use of computer systems becomes more widespread, there is a growing need for the ability to work with and on remote systems in an open and distributed environment. This requires more than just a network connection between systems. There is a need for transparency across the systems as well as a need for system services and users to be able to determine for themselves how they use and share their services within these distributed systems. This latter need means that whilst there has to be a strong mechanism for these services to protect themselves from malicious use, this very mechanism must be flexible and independent from as much of the systems' infrastructure as possible to operate successfully over heterogeneous systems. The properties of public key encryption and cascading proxies are combined in this paper, to develop a new mechanism which can provide a solution to some of these requirements.

The rest of this section describes the three areas of protection that are going to be covered. Section 2 reviews existing methods. Section 3 extends the principles currently employed to develop the Self Authenticating Proxy. Section 4 describes the actual SAProxy mechanism and how it works. Section 5 provides suggestions for the application of SAProxies and section 6 compares SAProxies with currently used mechanisms and highlights both the benefits and the price of the new approach.

1.1. Protection

In the past authentication and access control have been seen as the only two sides to protection that are required to ensure proper use of and access to services. These two functions have also usually been implemented as separate mechanisms. The discussion in this paper will suggest that the two mechanisms can be approached in a unified manner and that resource management is a third and important aspect of protection and one which may be easily incorporated within this new approach.

Access control (Authorisation) - Most service providers will only provide a service to those principals who have the authority to use such a service. The service provider must have a mechanism for checking whether or not a principal has permission to request an operation.

Authentication - Checking whether a principal requesting an operation is entitled to do so, access control, is pretty useless if the service provider is not really sure whether the requestor is genuine or an impostor. Authentication is the process by which principals can obtain a high level of assurance that the identity of the principal they are communicating with is genuine.

Accounting - Resource managers must ensure that use of a resource is not given to someone who has not got the funds, even if he has the right, to use it. Having provided a service the recipient needs to be charged for it. It is important both that the right person and the right amount is charged.

Authentication and access control have, up to now, been the heart of service protection. There has not been very much research, with the exception of covert channel analysis, that views resource control as a part of protection and as such the most commonly used protection methods only deal with authentication of the principals involved and access control.

2. Existing Authentication and Access Control Methods

Authentication of principals is most usually implemented using a shared key encryption algorithm. There is an authentication server, in each management domain, which shares a secret key with each principal. The authentication server is called upon to verify the principals and it does this by checking that they have knowledge of the shared key.

Access control is usually provided either by Access Control Lists (ACL) or by capabilities. These methods represent the same access rights from either the service provider's point of view or the client's point of view. A service provider may have an ACL which identifies which clients have rights to use the service and what those rights are. A capability is a 'token' held by a client which identifies a service and the rights to that service. The mechanism proposed in this paper is developed from capabilities and not ACLs, so the next section describes how capabilities have been used and extended.

2.1. Capabilities and Authentication

The traditional or 'pure' **capability** consists of the identifier of the service(s) and the access rights for that service. Being in possession of a capability is sufficient to grant access to the service. Problems arise with **modification, replication, revocation and theft** of capabilities. Capability systems have been implemented using either hardware support or system software to help prevent these problems. This proves to be very restrictive and problematic in a distributed heterogeneous environment. Modification can be prevented, or at least detected, if the capability is protected by an encrypted checksum e.g. Amoeba [Mull85]. Amoeba uses capabilities which are protected by signatures provided either by One Way Function boxes or public key encryption. There is no verification of a user's identity although Amoeba does provide a mechanism whereby a user can register a unique 'signature' with the system and this signature is then used to influence the capability checksum. The mechanism proposed in Amoeba is dependent on all the distributed systems running an 'unviolated' copy of the Amoeba kernel.

Capabilities can be allowed to propagate freely and then checked against the user's access rights at the point of access, as suggested by Karger [Kar88] in the SCAP architecture This is costly in performance and restrictive in that access is checked twice and is dependent on the policy checker not the capability.

Replication and theft can be prevented by making capabilities identity dependent as in **ICAP** [Gong89], Here, the identity of the user entitled to use the capability is embedded in the capability and protected from modification by the checksum. This makes replication and propagation of no use, but the user of the capability has to authenticate himself to prove he is the real owner when the capability is presented. The authentication mechanism uses shared key encryption and an authentication server.

This dependence on an authentication server is present in most of the other capability type methods currently in use e.g Kerberos [StNeSc88]. This is a trusted third party authentication service based on the Needham Schroeder model [NeSc78] which produces, for each user, a 'ticket' which gives the user authority to use particular services. The security relies on the authentication servers but not on the user end systems although the integrity of the authentication server itself does rely upon that of the system in which it resides. A development of the notion of a capability is the 'proxy', where a principal can act using another principal's authority, not his own. The authority has to be deliberately and verifiably 'granted'.

2.2. Proxies

The notion of delegating authority, as used by Sollins [Sollins88] with 'cascaded authentication', was further developed by Neuman [Neu91] , with 'proxies' and Bull et al [BuGoSo93] with 'access certificates'. A 'passport', as defined by Sollins, is a token which identifies the originator and is digitally signed at each transit point, so that each participating transit point is identifiable, thus cascading the authority. It also includes any constraints imposed at each transit point. The digital signature implies unforgeable identities of the originator and the principal entitled to use the passport. Although this scheme allows constraints to be forwarded, a passport is just an authentication token and is not sufficient to grant authority at the point of access. A passport originating from A may say that principal B can act on A's behalf within the constraints included in the passport.

Neuman looks at authentication and access control slightly differently. He says that an 'authentication server' need not specify that a principal has particular rights, instead the authentication server assumes all rights to the service it is protecting and issues a proxy giving another principal, the grantee, some or all of the server's access rights. A 'proxy' is a token that allows the grantee to operate with some of the privileges of the grantor. The proxy can be restricted to particular services and access rights. The grantee is the only principal entitled to use the proxy, and does so using the rights granted in the proxy. This principal may in turn act as an intermediary and may pass on some of his rights to another party in another proxy. The original proxy from the server has got to accompany the second proxy to show that the intermediary had been granted such access rights by the server in the first place. All grantees have to show that the proxy was obtained 'legally' and so at the point of access they have to prove that the grantor did indeed pass on those rights. This is done by proving possession of the grantor's credentials i.e. the grantor's secret. The grantor's secret key must also be given to the grantee. This complicates the procedure and requires a secure mechanism for passing on the key.

The mechanism proposed in this paper is very similar to, but developed independently from, the 'access certificate' mechanism which is also based on shared key encryption. Bull et al take a server-based view and assume that each service takes on responsibility for issuing authority to its customers. The server issues 'access certificates' which behave like capabilities and are created, issued and cryptographically signed by the server with the following differences; that the server can check itself as the originator, that the certificate is assigned to a specific principal and that server can check delegated signatures and check this information against a security policy. A certificate need not originate from the server itself but it may originate from a trusted entity provided the server has the means to check this authority against an ACL entry. Where a certificate is transferred to a recipient that has no ACL entry the delegation has to be confirmed by a signature. In all cases the server controls access even though reference to another authority may be needed to make a decision. This scheme also allows ACL entries to migrate throughout the distributed system as opposed to being maintained by the server i.e. the server may issue identity based capabilities for services offered. This view also differs from previous ones in that this model allows a high degree of autonomy so that servers themselves may migrate without concern about the system infrastructure. It does however imply that a client has to have complete trust in the behaviour of the server.

The three models described in this section begin to place the onus of responsibility for the access and use of services on to the entities involved in a client/ server transaction, and remove it from the system infrastructure. This makes operating in a distributed heterogeneous system more flexible than in the past. However, they are all designed around shared key encryption and an authentication server. This in itself restricts the domain within which a user can be authenticated to that of the authentication server the user shares its key with i.e. the user is still tied to a particular system and also authentication servers can masquerade as clients. Although all three schemes can operate using public key encryption, this is not developed and the properties of such a scheme are not exploited. The scheme proposed in this paper makes use of the features available to make the areas of trust as local to the principal as possible and to overcome the difficulties of being tied to an authentication server in a particular domain.

3. Extending Proxies to SAProxies

Proxies provide a means for passing authority from one principal to another. Public key encryption provides each principal with the means of producing a unique signature. By combining the properties of public key encryption with cascading proxies, a mechanism can be developed which enables a principal in authority to pass on some (or all) of his authority to any other principal and for the second party to prove beyond reasonable doubt, that this delegation of authority was intended. This is achieved by generating a proxy to pass on rights to another principal and signing this proxy with the grantor's secret key. A cascade of proxies is created when a principal who has a proxy that grants him some authority, delegates all or part of this authority to yet another principal by generating a proxy for that second principal. Thus, the grantee becomes a grantor, and passes his proxy (proof that he has authority) and the delegatee's proxy (proof of delegation) to the delegatee.

A 'signature' is a cryptographic checksum which is dependent both on the data it signs and on the secret key used. An invalid checksum shows either that the data has been tampered with or that the secret key used to produce the checksum does not match the public key used to check it. Thus if each principal has his own secret key then each signature is unique to that principal i.e.

unforgeable. The public key for each principal is tied to that principal's name by a Certification Authority (CA). The CA is what provides each principal with an unforgeable means of identification - a Public Key Certificate (PKC). Each principal, including the CA, has a public and secret key pair. A PKC for any principal is that principal's name and his public key 'signed' by the CA's secret key. Any principal, local or remote, can verify the signature on a PKC by using the CA's public key, A principal can prove his identity by producing a signature with the secret key that can be verified with the public key in his PKC. Knowledge of the CA's public key is easily obtained because this is made globally public so that it is both difficult for an impostor to change it and easy for any one to use it.

A principal's PKC, therefore is his unforgeable identity. By presenting this PKC with any statement signed with the matching secret key, anyone can verify that only the principal identified by that PKC could have made that statement. In the same way the proxy issuer, the grantor, can prepare a proxy delegating some of his authority to another principal, the grantee. Instead of just including the grantor's name in the proxy, the grantor's PKC (which has the grantor's name) is included. The proxy is signed by the grantor's matching secret key. As the proxy includes the grantor's PKC, then the proxy becomes 'self-authenticating' - a SAProxy. The SAProxy can be verified by any principal because it holds all the information needed to do this. The benefits are twofold in that there is no need to call an authentication server either to generate these credentials or to verify them as both can be done 'locally'. Revocation of SAProxies, as with capabilities, is not simple, but there are mechanisms which can be used if a unique number is also included in the SAProxy which is discussed later. To summarise then,

A Self Authenticating proxy (SAProxy) consists of:

- the grantor's public key certificate .... *identifies the grantor and his public key*
- the grantee's name .... *name of the principal entitled to use the SAProxy*
- the service identifier/ name.. *the address of server*
- the rights allowed to the grantee .... *the authority for the grantee*
- the signature block .... *provides authentication of the grantor and tamper detection*

The SAProxy is signed with the grantor's secret key.

The grantor's PKC is an unforgeable entity on its own and as such does not need to be covered by the overall SAProxy signature to ensure its integrity. However the name of the grantor in the PKC has to be 'tied' in with the rest of the SAProxy in an unforgeable manner. To do this, the PKC signature block is included in the data that is signed by the grantor to produce the SAProxy's signature.

The issuer of the SAProxy can have his identity verified because of his PKC - this is essentially his 'passport,' and is used to verify the signature of the self-authenticating SAProxy. This is the same as sending a 'passport' with a document that is signed by that passport's owner so that the signature can be verified. The strength of the authentication of that signature lies in the belief that the passport is genuine. The PKC can always be verified using the CA's public key.

Similarly, when the SAProxy grantee uses the SAProxy, he has to provide his own 'passport' (PKC) so that his identity and signature can be verified. When the grantee presents a request using the grantor's rights, he must have signed his request with his secret key. The grantee also presents his PKC, passport, as proof of a 'real identity'. The principal verifying the request verifies that the 'passport' is genuine and does indeed 'belong' to the principal named as grantee in the SAProxy. The grantee's signature over the request is then verified using the public key in his PKC.

The SAProxies provide authentication of the entities that present them without the need for an authentication server, and they are independent from the system infrastructure. These features make SAProxies particularly suitable for use in a distributed heterogeneous environment where users are working in geographically different locations, services are provided by distributed servers and there is no universally trusted system kernel.

3.1. Using SAProxies

As SAProxies are independent from the system infrastructure, they lend themselves to be used quite easily by services offered to clients. The security policy decision makers and authorisers do not have to be special servers, such as authentication servers, but the service providers can themselves set and control their own policy decisions and implementations. Alternatively, a user (or group of users) who do not trust the service provider completely, can insist that any transactions carried out by that service has been vetted by an entity that the user trusts. Proof that the trusted entity participated in the transaction can be provided in a SAProxy from that entity.

Services can become responsible for their own authentication and access control. The service providers become the SAProxy grantors controlling to whom they give access and what permissions they get. They can maintain their own security policy and check this when they issue SAProxies or they may refer to a separate policy checker, but essentially they are responsible for any SAProxies they grant. A separate policy checker may provide proof that it has endorsed the decision by providing a token that can be included in the cascade of SAProxies e.g. another 'SAProxy'.

SAProxies can also be used as a means for managing resources. As well as holding access rights, SAProxies may hold resource funds and so resource managers may use SAProxies to control the use of their resources, and because of the fact that they can always be attributed to one principal, they can be used for accounting purposes.

3.2. SAProxies and Role Holders

SAProxies can be used like capabilities in the sense that they are generated and given to the grantees and the issuer does not **need** to keep a record of the SAProxies issued. (The issuer will always know if he issued them or not by checking their signature). Storing the SAProxy is then the responsibility of the grantee. Therefore there is no demand for the use of ACLs although SAProxies and ACLs are not mutually exclusive.

In the area of documents, for example, there may be a group of users that jointly 'own' a document. There is a distinction between giving a user the authority to use your permission and giving the user some access to your document in his own right.

The first is an example where user A, who has been granted authority by the document server, S, may delegate this authority to another user, B, by generating a new SAProxy and creating a cascade signed under A's secret key. The user B can only operate using A's rights if the cascade includes A's SAProxy to give B authority.

S —> A	S sends A a SAProxy granting A authority to access the service
A —> B	A send B a cascade of two SAProxies, allowing B to use A's authority to access S.

B —> S B sends a request for an operation to S, with the cascade of two SAProxies, using A's authority.

The second situation is where A authorises access rights for another user B, whereby, B then becomes a new member of that group, and has authority in his own right though not necessarily with the same permissions as any other member of the group.

S —> A S sends A a SAProxy granting A authority to access the service

A —> S A asks the server to grant B authority. A's SAProxy must give him the right to make such a request.

S —> B S sends B a SAProxy granting B authority to access S.

B —> S B sends a request for an operation to S, with his SAProxy, using B's own authority.

This can be seen as the difference between 'delegating authority' and 'issuing authority', whereby the first gives the grantee access only if accompanied by the grantor's authority i.e. SAProxy, and the second gives the user authority in his own right.

A group of 'owner' users can be seen as particular role holders. This is particularly true in the area of Computer Supported Cooperative Working, where several principals are responsible for the development of a document [GuAbCa91]. The SAProxy grantor e.g. the document server, issues a SAProxy with an 'owner' role or access right, to those users who are members of the owner group. This 'right' gives them owner status. Delegating authority is done by cascading proxies, but issuing authority (as defined above i.e. adding a new member to the group) requires the corporation of the entity responsible for granting principals authority in their own right[1]. Any group member may request that another user join the group. This will result in the server generating a role holder SAProxy for the new member.

An alternative to defining role holders in a SAProxy, is to represent them in an ACL. The 'group members' will then generate their own SAProxies when they request an operation and their authority is checked against their ACL entry [Sollins88], [BuGoSo93]. These users may then delegate authority to other users by issuing them with a SAProxy they have signed without needing to modify the ACL. They may also introduce a new member of the group, 'delegate' some rights, by requesting an entry in the ACL for the new user. This request is authorised by their SAProxy and the rights in the ACL. SAProxies therefore can be used as the mechanism for updating the ACL. SAProxies lend themselves naturally to a distributed service as the role holder SAProxies can be verified by any one of the servers whereas any ACL modifications will have to be communicated to all the servers.

Revocation of a SAProxy obviously becomes more complex the more distributed the system. Keeping a list of SAProxy revocation numbers may become necessary, but this is not sufficient for distributed services, as all the servers of the service have to keep the revocation list up to date. This is similar to the problems with 'hot-card list' and ATMs (Automated Teller Machines). It may be that a central list of revoked SAProxies is kept and transmitted to the servers of a service

[1] Essentially this is the service provider, although this may call on a separate entity to actually issue the SAProxy.

either when the list is updated or at regular intervals.

This view of role holder SAProxies can also be used in resource management as particular role holders can be used as resource managers who are held ultimately accountable for the use of the resource.

Having described how SAProxies may be used, the following section describes the basic mechanism required to handle SAProxies.

4. SAProxy Methods

This section describes the facilities required by the entities using SAProxies and the actual mechanism itself. Most of the examples refer to SAProxies for access control only and not resource control. This is only for simplicity and the mechanism may be applied in the same manner to resource control as to access control.

4.1. Basic Requirements

Each principal has a public and secret key pair (for principal S, these are S_{K+} and S_{K-}) and access to a digital signature service. To ensure that a principal's public key is 'uniquely' and 'securely' tied with that principal's identity, the public keys must be certified by a trusted CA or a set of trusted (by some) CAs. The CA runs off line and requires that its administrators guarantee the identity of each principal presenting a key for certification - more importantly that an impostor cannot certify a bogus new key for another principal. A CA produces a public key certificate for each principal (for S, S_C) signed under the CA's secret key CA_{K-} (which must not be compromised).

A public key certificate for a principal, e.g. server S, consists of the following:

S_C: CA_{id}, S, S_{K+},S_{Csig} where

CA_{id} is the identity of the CA, S is the principal's identity, S_{K+} is S's public key and

S_{Csig}: $\{CA_{id}, S, S_{K+}\}CA_{K-}$ is the CA's signature of that PKC.

The CA's public key, CA_{K+}, is made sufficiently public (plastered everywhere) so that anyone verifying a key certificate will know it and so that it is very difficult for an impostor to pass off another key as the CA's public key. Note that a principal may himself issue a certificate for the CA's public key. This certificate allows the principal to later verify the CA's key without having to remember its value, not trust somebody else to provide the right value.

If a principal has a certified public key for another principal, then anything signed by the matching public key pair authenticates that principal, is tamper detectable and unforgeable because no-one else has knowledge of the corresponding secret key.

4.2. The Mechanism

A SAProxy originates from a service provider[2] and is signed by the provider's secret key. A SAProxy is generated for a user who is allowed to use the service. The service provider, S, generates a SAProxy, A_P, for client A.

A_P: S_C, A, I, ar, A_{Psig} where

[2] The SAProxy does not have to be directly generated by the service, it may call upon another entity to do so. The point is that it is generated at the request of and under the responsibility of the service provider.

I is the service to be accessed and 'ar' are the access rights and

A_{Psig} : $\{S_{Csig}, A, I, ar\}S_{K-}$ is the server's signature of the SAProxy.

The SAProxy grantor, the service provider/ administrator, signs the SAProxy with its secret key S_{K-}. The identity of the client to whom the SAProxy is given, A the grantee, is embedded in the SAProxy.

When the client, A, requests an operation of the service e.g. READ, it presents the request with this SAProxy, signed with the client's own secret key, A_{K-}.

A-request: A_C, A_P, READ, I, $\{A_{Csig}, A_{Psig}, \text{READ}, I\}A_{K-}$.

This client's public key certificate, A_C, is included in the request. By having the signature blocks for the client's PKC, A_{Csig}, and the SAProxy, A_{Psig}, included in the data signed by the client the requests and the authorising tokens are all bound together.

The service protection checker performs two functions. It **authenticates** the client presenting the request as a recognised certified user, and **authorises** the operation requested by that user; it must check that that client has permission to request that operation.

The protection checker first of all verifies the client's PKC, A_C, to establish the client's identity.

$\{CA_{id}, A, A_{K+}\}CA_{K+} = A_{Csig}$[3]

It then checks the signature of the request received with that public key. This shows that it was indeed that client who presented the request (only he knows the matching secret key).

$\{A_{Csig}, A_{Psig}, \text{READ}, I\}A_{K+} = \{A_{Csig}, A_{Psig}, \text{READ}, I\}A_{K-}$

Having got this far, the protection checker verifies that the SAProxy, A_P, is valid by checking it with the grantor's (server) public key, S_{K+}.

$\{S_{Csig}, A, I, ar\}S_{K+} = A_{Psig}$

The identity of the client is checked against the grantee named in the SAProxy; they must be the same, in this case A.

The authority of the client is verified by checking the access requested against that allowed in the SAProxy, i.e. is READ in the 'ar' field in the SAProxy.

The service is then presented with the result i.e. the client can/ or cannot perform the operation. It is then up to the service to perform the operation depending on the policy it is implementing. It is possible that the protection checker can be generic and trusted by the server, but not necessarily a part of the server. In this case the protection checker is then just a protection checking service available to other services.

Any client that has been granted authority can in his turn delegate all or part of this authority to another principal. The mechanism allows authority to be cascaded from one principal to another whilst maintaining the ability to verify each delegation and authenticate each principal involved in the cascade.

[3] The '=' does not denote an algebraic equivalent, but is being used within this notation to represent the notion that a signature block produced using a secret key can be verified by applying the public key over the same text and signature block.

4.3. Delegation of Authority

One client may allow another user to request some operations form the service, S, on his behalf using the client's own rights and not those owned by the delegate user.

Client A delegates to user B, by 'cascading' his authority. A gives B a cascade of two SAProxies, A's own SAProxy, A_P as above, and a SAProxy, B_P, generated by A naming B as the grantee. This SAProxy may have restricted rights if necessary.

B_P: A_P, A_C, B, I, *ar*, B_{Psig} where

'*ar*' are the 'restricted' access rights and

B_{Psig} : $\{A_{Psig}, A_{Csig}, B, I, ar\}A_{K-}$

A's PKC, A_C, and SAProxy, A_P, are bound into B's SAProxy by including their signature blocks in the data A signs. B then presents his request with this cascade all signed under his secret key, B_{K-}, to the server.

B-request: B_C, B_P, READ, I, $\{B_{Csig}, B_{Psig}, READ, I\}B_{K-}$.

B can delegate to another user C in the same way as A, augmenting the cascade of SAProxies from one user to another. Cascades may be restricted if SAProxies have delegation rights - any user in the cascade may restrict this right to the next user and thereby stop the chain i.e. a non-transferable SAProxies, the protection checker then checks the 'proxy rights' to see if the grantee has authority to pass rights to another user.

B_C and the request from B is verified as in the previous example for A. Then B_{Psig} is checked using A_{K+}. Assuming B_{Psig} is valid and that B in the SAProxy is the same B that presented the request then the second SAProxy, A_P, is verified. This is repeated for all SAProxies in the cascade.

The notion of cascading SAProxies to delegate authority may also be applied to represent combined authority to request an operation.

4.4. Combined Responsibility

In this situation, SAProxies are not necessarily 'cascaded' but they may be chained to indicate that several principals are taking joint responsibility for an action.. Thus if it takes two role holders to perform an action these can be represented by a SAProxy from each of them in a chain. The service provider generates a SAProxy for each principal A and B.

A_P: S_C, A, I,ar, A_{Psig} and B_P: S_C, B, I,ar, B_{Psig}

If user A sets up a request for an operation, he includes his SAProxy in the request and signs it all with his secret key.

A-request: A_C, A_P, READ, I, A_{reqsig} where

A_{reqsig}: $\{A_{Csig}, A_{Psig}, READ, I\}A_{K-}$.

This is then passed to the second principal, B, who, if he agrees to the transaction, at a minimum just has to add his SAProxy and sign the whole request with his own secret key.

B-request: A_C, A_P, READ, I, A_{reqsig} , B_C, B_P, $\{B_{Csig}, B_{Psig}, A_{reqsig}\}$ B_{K-}.

This shows that indeed both the principals have seen the request and both have assented to it. It is then up to the service provider to check that he has received the correct authorisation by checking against its security policy. This applies accurately to serial authority but does not appear to reflect parallel authority. However, the only difference between two events having to occur serially or in parallel is that in the first case it matters which happened first, and in the second it doesn't, all that matters is that both happened. It seems that both can be reflected in the structure of a chain of SAProxy.

The repeated use of a cascade of SAProxies may be a problem, particularly with resource control, as a user who has rights to a resource must also have the funds for using it, so it may be necessary to limit the use. This is closely related to revocation or making a SAProxy invalid.

4.5. Revocation

Revocation can be achieved by including a unique number in every SAProxy. Any entity that issues a SAProxy, keeps a list of these numbers, then when a SAProxy is to be revoked, the protection checker is given the unique number of that SAProxy[4]. The protection methods can check the SAProxy against this list of numbers as part of verifying its validity.

This has the disadvantage that every principal issuing a SAProxy must keep tabs of the revocation numbers it has issued and who they were issued to. It also implies that the service provider is not really stateless and the distributed servers have to keep their lists up to date, as mentioned previously as a SAProxy revoked on one system must also be invalid in another.

Each principal that is involved in a cascade of SAProxies, adds his own SAProxy to the cascade. Each SAProxy has its own unique number and so the intermediary in the chain can also inform the end-server that such a cascade is to be revoked.

SAProxies can also carry an 'life-span' so that the same request can only be used a specified number of times within that life. The server can keep a table of SAProxies which are 'alive' and restrict their use to that number of times within that 'life'. A general solution to revocation of SAProxies and restricting the number of times each is not proposed in this paper, but it is possible that solutions will be different and dependent on each type of application.

The next section discusses the applicability of SAProxies using the mechanism as described above.

5. Application of SAProxies

This section describes how SAProxies may be used to provide protection and resource control of services and resources.

5.1. User authentication and access control

It is easy to see that SAProxies provide a mechanism which a service can use to confine the operations it performs, to those requested by authentic and authorised users, with a high level of assurance. Similarly, a user may also want proof that the transactions are confined, and that an outsider will not be able to violate the user's trust in the system. A first step towards this is for servers to keep an audit trail of their actions and the authority under which they performed those actions. The user of the service can then check that all operations performed have been properly authorised and can hold the service accountable for any violation of trust.

The service relies on its own resources and not on the system infrastructure, and its integrity is not weakened by any weak element in the distributed system. User authentication and access control

[4] The protection checker essentially keeps a revocation list.

are combined within one mechanism. The use of public key encryption means that user requests and SAProxies are 'self authenticating', there is no need to use a trusted authentication server - all that is needed is a local public key algorithm function. This reduces the amount of trust that both the server and users have to place outside themselves and reduces the number of messages in any transaction. This must be balanced against the more 'expensive' form of public key encryption. The SAProxies include each grantor's public key certificate, so there is no need to access a database of public keys either.

As has already been shown, it is possible to represent the need for joint authorisation as well as delegating authority to another user, within a cascade of SAProxies. SAProxies allow for a group of principals to be defined as role holders. The SAProxies can include a role holder identity, or the role of a principal can be represented by the rights he has in his SAProxy. Particularly, a group of users can be set up as the service administrators i.e. the service provider issues these principals with role holder SAProxies which identify them as service administrators. These administrators may be distributed throughout the system. These administrators are then responsible for issuing SAProxies to the clients of the service, who may in turn, delegate their authority to others. All requests to the server must include one or more of these administrator SAProxies - the number of administrator SAProxies required being determined by the security policy in use.

The service provider does not deal with the users of the service, and leaves granting authority to its clients to the administrators. If the administrators are trusted by the service provider, and they are the policy implementors, then this may result in a generic server which is not concerned with the integrity of requests, but leaves this chore to its administrators. The decisions for granting access to the service can then be made locally by each administrator thus reducing the amount of work and the extent of user knowledge required by a single service which is serving users on several systems.

The SAProxy mechanism eliminates the difficulties in cross-domain authentication when a principal wants to use a service in a domain different to his own. Each domain has its own CA, and there is no need for a hierarchy of CAs across domains as there is with authentication servers. This is because each CAs' public key is publicly known, and so a CA in one domain can verify any PKC certified by a CA in any other domain - all that is needed is for the CAs to be able to obtain each other's public keys (as these are globally public there is no problem). Therefore, the independence of the CAs enables cross management domain operations to be carried out without the need for trust and secure pair-wise communications between each domain's CA. There is also no problem with denial of service if an authentication server becomes unavailable because there is no on-line access to a particular system server.

The SAProxy scheme is particularly suited to distributed services. If the service is distributed, then any one of the servers can verify another server's SAProxy and check that the request is within the constraints imposed by the access rights as stated within the SAProxy before continuing with the operation[5]. Servers providing the same service on different systems have no trouble honouring requests based on the authority of another server as they can always verify each others SAProxies and PKCs. This does require a revocation mechanism that informs all the

[5] If there is only one server and it is the only one who needs to check this SAProxy, it does not need to use public key encryption, but it may use its own proprietary signature algorithm/ hash function.

servers when a SAProxy is revoked on any one system. Having discussed the role SAProxies can play in providing a protection mechanism over a distributed system, the next section identifies the way in which the same mechanism can be adapted to provide resource control in the same environment.

5.2. Resource Management

All services can be viewed as resources and as such users need to be allocated 'an amount' of the resource to be available to them and then must be charged for what they use. 'The use of a resource cannot be authorised unless funds are available and funds cannot be made available to a user until that user has been authorised' [Neu91]. From this statement it is obvious that resource allocation is closely related to authorisation. Account servers usually maintain, for each user, a unique identifier, an ACL, and a collection of records specifying the various 'currencies' a user has available e.g. file blocks, printer sheets and CPU time.

The SAProxy mechanism can be used to manage these resources. A resource manager can issue SAProxies with the amount of that resource available to the user embedded in that SAProxy. The user entitled to use a service, such as a printer, may only use so much of it, say only 50 sheets. Therefore, the SAProxy for that service would have the right to print with a limit of 50 sheets. The service provider can then ensure that a user request does not exceed the limit in the SAProxy. A user that has access to a resource may delegate authority to a principal that is not totally trusted, secure in the knowledge that if he restricts the amount of resource available to that user then there is only a limited amount of damage he can do. One of the problems this gives rise to is how to prevent repeated use of the SAProxy. SAProxies need to be uniquely identified c.f. cheque number, and given a life time so that their use can be restricted e.g. they can only be used once in their life time.

A server may manage its own resources by keeping an accounts manager's records itself but it may also have an accounts administrator whose authority is required before carrying out an operation. The accounts manager has to keep a record for each user who has authority in his own right to make use of its service. This record states the amount of the resource available to the user. When a request is received, the server checks the amount authorised in the SAProxy against the amount available and if there are sufficient funds the request can progress. When resources are consumed, the accounts record for that user can be modified.

A server that uses service administrators can delegate the task of resource manager with responsibility for sharing the resource fairly to one of these service administrators. In this event, all requests to the server must have a SAProxy from this administrator in the cascade[6]. This SAProxy authorises the user to consume a stated amount of the resource. Alternatively, a server can allocate the funds available between all the service administrators and then they become responsible for sharing their funds with the users they manage.

It is always the user who has the authority in his own right, as identified by his access or role, who is charged for the service, even though he may have delegated authority to another principal. It is then up to the grantor to settle with whoever put in the request in his name if necessary. All actions can be confidently attributed to those involved as all requests are signed by each principal's secret key.

[6] This is an example of combined authority required to perform an action.

When an operation has been completed the service provider must charge the user (his system) for the use of the service. In a distributed system this will require firm proof that the service was indeed requested and supplied as there may not be much trust between the systems.

A verifiable log of the request is available if an audit trail of requests and the authorising SAProxies is kept, and sent to the user's system. This may be copied to all users in the cascade as well. This proves that the request was made. The system of the grantor, the user who will be charged, must now be assured that the service was supplied. SAProxies can again be used to claim payment.

The following two approaches can deal with this requirement. The grantor can tell his system of the service and ensure that his system recognises it as a valid service, or a resource administrator could issue each user with SAProxies of certain value, voucher SAProxies, that they can spend as they wish. These voucher SAProxies may be for any service, or for a particular service, tied to a particular user identity or not. If the system recognises the server, then it can authorise it to claim for services rendered by issuing it with a SAProxy signed by the system's 'paymaster'. This SAProxy guarantees payment on proof of use of service. The paymaster's SAProxy may have a limit on the amount that it is prepared to pay and the service's resource manager must consult this before authorising the request. When the demand for payment is sent, this SAProxy and the log of the operation must be a part of the invoice which is signed by the server's secret key so that it cannot be forged.

If users of the service are issued with voucher SAProxies, then one of these SAProxies must accompany the initial requests for service, so that the server's resource manager can have some assurance that the service will be paid for. When charging for the service, this voucher SAProxy will, as above, form a part of the server's signed demand for payment.

6. Finally

This paper has described the development of the SAProxy scheme and how it can be applied in an open, distributed and heterogeneous environment to provide protection and resource management. The following two sections summarise the benefits and discuss further issues.

6.1. Summary

The benefits of SAProxies arise from the use of public key cryptography. The process of binding the PKC to a proxy gives it the self-authenticating property. The fact that a public key environment gives each principal his own secret key, known only to him and yet verifiable by the rest of the world is what gives SAProxies their independence from any particular security domain or authentication server and makes them extremely suitable for an open environment. A SAProxy authorises a principal to access a service by embedding the access rights within the SAProxy. Authentication of the user presenting the request authorised by the SAProxy can be verified because the request is signed by that user's secret key which matches the public key in his PKC. Delegation of authority and combined authority can be verifiably represented within the mechanism. Attribution of actions performed on presenting SAProxies is possible because each principal's secret key is not shared with any other and so any request signed by a principal's secret key can be attributed to that principal. Tampering with the SAProxy and requests can be detected because the signature is dependent on all the bits of the data signed. Where operations performed are logged together with the SAProxies that authorised them, then an audit trail of actions which makes both the servers and clients accountable, is available. If

SAProxies are extended to include funds allocation, they adapt easily to enable resource management.

The approach proposed in this paper does not need an authentication server or an equivalent. The function of the CA is quite different - this function is to generate a PKC for a principal and is performed off-line and only once[7] per principal. Generation and verification of a SAProxy supporting a request may be done locally by any principal or service. Generation is done by a principal or service using his own secret key in some environment that he alone needs to trust e.g. smart card, local PC or local system facilities. Public key SAProxies are 'self-authenticating'; they share no secrets and carry all the information they need to be verified. All that is required is a local copy of the public key algorithm, which is not secret and can exist anywhere, and the relevant PKCs. This can be compared with existing systems that need on-line access to a trusted and trustworthy authentication server just to authenticate the principal. Therefore, there is no need for any communications with this separate server, which might become a bottle neck and cause denial of service because of lack of availability i.e. SAProxies have no dependence on any system services or infrastructure. Furthermore, there is no restriction on the domain in which the SAProxy is verified because as it does not need access to any domain server at all, it can be done anywhere. All that is needed is a copy of the public key of the CA which generated the PKC in the SAProxy.

6.2. Discussion

The approach proposed in this paper and the trustworthiness of the SAProxy scheme described is dependent on the following factors which may prove to be possible weaknesses. These factors are the public algorithm used and its strength, the integrity of the CAs and generating key pairs and maintaining the secret keys for the principals.

The public key algorithm used must be strong, as must the keys generated. This applies to the CA's key pair particularly. The longevity of the public keys (particularly the CAs') has to be determined i.e. how long before the secret key can be determined from text and the public key.

The security of all the CAs' secret keys must be maintained. If a CA's secret key is compromised then anyone who has knowledge of it can certify any public keys, including replacing a valid user's public key with a bogus key, but this cannot be done without leaving evidence; until the foul play is detected all actions are attributable to that user. Management of the CAs must be rigourous. The CA administrators must be trustworthy and the certification procedures stringent. The relationship between CAs in different management domains must be clearly defined so that violation of a CA in one domain does not make users in another domain vulnerable to attack.

The third factor is the ability of the principals to keep their secret keys secret. This sounds highly unrealistic but it is based on the reasoning that it benefits all principals to keep their own secret key secure and that if a principal's secret key is compromised, then the impostor can do no more than that principal is entitled to do and that no one else will be attributed with that action other than the user whose key was compromised.

There are also practical factors which need to be considered. The computational requirements

[7] A principal can of course generate a new public key pair whenever he wants, but within any given set of transactions a PKC need only be generated once.

and performance costs of using a public key algorithm such as RSA must be evaluated. The extra lengths of data in each transaction, generated by the accompanying SAProxies and signature blocks needs to be assessed. Whereas this may be an unacceptable overhead in short requests, in other areas such as in document development, this may be negligible. These two factors need to be weighed against the flexibility of the scheme, the reduction in the number of transactions because there is no need to continually access authentication servers (especially across domains) and the reduction in the amount of trust principals have to place in remote system's authentication servers. Another practical problem is key management. The availability of a key generation mechanism for all the principals and the difficulties that arise when changing a CA's key pair are of practical rather than research interest, yet they often pose the biggest problems when trying to implement similar schemes in a real life situation.

Further work remains in determining the range of applicability of SAProxies. Some of the areas that SAProxies may be applied to may be decided by answering the following questions. Can SAProxies be used for fine grained protection of objects, can they be a means of ensuring fair scheduling of processors, can they be beneficial in an environment which has greater security demands than an open system such as in military systems and can they be used to enforce mandatory access control and the Bell and LaPadula security and Biba integrity models.

Although the above issues still have to be resolved and the extent of the use of SAProxies determined, the resulting mechanism appears to solve many of the difficulties encountered by comparable schemes as well as providing a flexible and comprehensive means to protection and resource management.

6.3. Conclusion

In this paper, we have shown that by combining the properties of public key encryption with cascading proxies a single mechanism, SAProxies, can provide a unified and consistent approach to authentication, access control and resource management. SAProxies are more suited to an open and heterogeneous environment than other comparable systems because they exploit the properties (independence from trusted server, localised trust, local verification) of public key encryption. One of the major benefits of using public keys is that there is no need for a single trusted server and so users are not restricted to operating within any domain.

Schemes that are dependent on an authentication server require that the server is trusted by all the principals it serves and, as it holds each principals' secret key, it must keep the keys securely and must behave correctly. Should such a server become malicious then all principals who share their secret keys with it are compromised. If a CA is violated and its secret key compromised, the worst that can happen is that 'false' PKCs can be generated for genuine users. This means that an impostor can request an operation under another principals authority using the 'false' PKC. However, a principal's secret key is not actually compromised when the CA is violated, and so the principal can show, by proving knowledge of his own secret key, which is his genuine PKC. The principal can therefore prove retrospectively what actions he has authorised and which he has not, and this is not possible with shared key authentication.

Finally, SAProxies show a way of putting together three elements of protection, authentication, access control and resource management, which up to now have always been viewed as totally separate mechanisms. Funding information is easily incorporated in a SAProxy which may be used by resource administrators. Funding SAProxies can be checked at any time and do not require that the resource managers have a relationship with an authentication server. A resource administrator may be consulted before a service is provided (or it may volunteer a SAProxy at some stage in the cascade) and grant its authority in a SAProxy.

In general, SAProxies provide a consistent approach and mechanism to authentication, access control, delegation of authority and resource management. The flexibility of the mechanism and its independence from the system infrastructure enables the integrity of work to be maintained in an open and heterogeneous environment.

References

[BuGoSo93] Bull J., Gong Li.,Sollins K., 1992. Towards Security in an Open System Federation. In: European Symposium for Research in Computer Security ESORICS '92. Toulouse, France, 23 Nov 1992. Springer-Verlag Lecture Notes in Computer Science.

[Gong89] Gong, L., 1990, Cryptographic Protocols for Distributed Systems. Cambridge: Jesus College. Thesis (PhD).

[GuAbCa91] Guan S., Abdel-Wahab H., Calingaert P., 1991, Jointly-Owned Objects for Collaboration: Operating System Support and Protection Model. Journal of Systems and Software. Volume 16, No. 2, pp85-95.

[Kar88] Karger, P.A., 1988, Improving Security and Performance for Capability Systems. Cambridge: Computer Laboratory. Thesis (PhD).

[Mull85] Mullender S.J., 1985, Principles of Distributed Operating System Design. Amsterdam: Vrije Universiteit te Amsterdam. Thesis (PhD).

[NeSc78] Needham R.M., Schroeder M.D., 1978. Using Encryption for Authentication in Large Networks of Computers. In: *Communications of the ACM 21 (12)* pp. 993-999. December 1978.

[Neu91] Neuman N.C., 1991, Proxy-Based Authorisation and Accounting for Distributed Systems. Seattle: University of Washington, Department of Computer Science and Engineering. Technical Report 91-02-01.

[Sollins88] Sollins K.R., 1988, Cascaded Authentication. In: Proceedings of the IEEE Symposium on Security and Privacy. Oakland, CA, USA. April 1988. IEEE Computer Society Press, Los Alamitos, CA, USA pp156-163

[StNeSc88] Steiner J.G., Neuman C., Schiller J.I., 1988. Kerberos: An Authentication Service for Open Network Systems. In: Proceedings of the USENIX Winter Conference 1988. Dallas, Texas. Feb 9-12, 1988.