DIVISION OF COMPUTER SCIENCE

Expressing a Policy

Marie Rose Low

Technical Report No. 266

July 1996

Expressing a Policy

Marie Rose Low

University of Hertfordshire

July 1996

Abstract

Policies, in particular security policies, are typically modelled using access control lists. This technique is not sufficient to model the full 'semantics' of a policy, especially when it applies to a task executed in a distributed environment. This article proposes a different means of expressing a policy based on extended capabilities.

1. Introduction

Most of our lives are governed by the policies under which we operate - it is our policy not to read another person's letters, to stop at traffic lights and so on. We apply these general policies automatically. In business, however, we have to adopt a much more formal approach. Policies are clearly stated and we are expected to follow them. This is particularly true of security polices in computer systems where access to resources and information must be restricted to authorised personnel. The enforcement of security policies is of great importance where tasks are carried out over distinct management domains each following its own working and security policy.

Security polices, or security requirements, are typically modelled using access control lists (ACL) or access matrices, in terms of principals, objects and access rights. However, ACLs are not sufficient to express the full semantics of a security policy. For example, they do not show any aspects of delegation; the authority granting access is not usually the same as the one which modifies the ACL. The use of Role Based Access Control (RBAC), where authority is granted to a role and then an individual is assigned to that role, is proving to be of great benefit when defining policy requirements.

This article clarifies some aspects of a security policy and discusses the tools necessary to express the policy, to check it and to enforce it. Firstly, we discuss different views of what a policy is, then we describe how the policies that governs two separate tasks are modelled and enforced. We then propose the use of a different technique, Self Authenticating Proxies (SAProxies) [7, 8], to model and enforce a security policy and show how this technique can be applied to the two tasks described. Finally, we suggest that this technique may be developed further into a language or notation with which to express the full requirements of a policy.

2. What is a Policy?

Lewis [6] claims that a policy is the attachment of a rule to a domain. Policy is often translated in computer systems into a set of rules. A policy can also be viewed as a set of obligations and constraints on a process that define the path the process must take. McClean [11] also suggests that a policy cannot just be stated as a set of rules, but that the rules are merely one specific solution that satisfies the policy.

In general, a policy describes the actions which may or must be taken by a principal when acting in a particular role. In [10], Lupu et al. define a role as "the authority, responsibility, functions and interactions associated with a position". Roles, in these terms, have rights and obligations. Rights are modelled as authorisation policies and obligations are modelled as obligation policies. Authorisation policies are defined in access control terms i.e. only certain principals have access to particular objects, and policies are typically enforced by implementing

ACLs. Defining an ACL for a role becomes the equivalent of stating a policy.

A straight forward ACL may be sufficient in a centralised system to express the security requirements, usually restricted to the constraints, where a policy checker is familiar with all the principals that may take part in an operation. However in a distributed environment we have to cater for different situations e.g. where authority to access an object in one system may be granted by a principal in another system. The policy must then cater for 'authorised principals not of this domain' and must be able to recognise these.

We suggest that policy is a set of obligations and constraints, but that these obligations and constraints can themselves be viewed as a set of rules. We define a security policy for a task, as the set of rules which state the constraints and the obligations that apply to each role that plays a part in that task. There is a need to formally specify a security policy in order for it to be validated and enforced. According to Boswell [2] "The formal validation of a formal security policy would ideally be a part of system development". Our aim is to develop a way of expressing these rules formally so that the full semantics of a security policy can be validated formally and verified and enforced in implementation.

3. Different Policies

The requirements of a security policy are best illustrated by looking at examples of different policies. The following examples show the formulation of two different policies and the proposed method for enforcing them. In these examples RBAC is used to simplify the definition and enforcement of the policy.

3.1 Preparing an Exam Paper

Coulouris and Dollimore in [4] examine the policy for setting an exam paper. This is a 'distributed task' - there are several distinct parties involved in the task. The principles behind the solution proposed in this paper are that participants in a task take on a role whilst working on that task. The actions that these participants are allowed to perform are constrained by the access rights granted to each role. A participant has to have authority to act in a particular role.

The roles that need to participate in this task are exam board members, Chair, examiners Ex1 and Ex2 and external examiner.

The roles have to perform the following functions:

- Ex1 creates a series of versions of the paper which are commented on by the Chair and other examiners,
- the examiners and members of the board need to be able to read the versions of the paper and comments,
- the chair and other examiners finalise the paper.

The task as described above is governed by the following policy in order to satisfy the requirements for a valid exam paper:

- no users other than those involved should see the state of a paper,
- papers are created and altered only by relevant roles and their delegates,
- one level delegation policy,
- a particular paper must be finalised by some set of the roles,
- delegates must be chosen from a specified group selected in advance by the Chair.

Coulouris and Dollimore [4] propose an implementation based on an access control list, a security template. There is a security template for each type of task and this template defines the rights each role has on each object used within that task.

3.2 A Medical Information System

In [1] Anderson proposes a security policy for a medical information system. This is a complex scenario which is reduced for the purposes of the security policy into the following participants: clinical professionals (under oath of confidentiality) and patients; the objects are patient records. Delegation by clinicians is not considered as the clinician is deemed to remain responsible for all such actions. In general a patient's record is available to all clinicians in a department/ surgery with the exception of a small part of the record which is confidential and only accessible by particular clinicians. The security policy has to consider the need to restrict access to part of a patient's record to specific clinicians. Patients must be notified of any third party that is given access to their records. This includes special notification when the clinician being added already has access to a large number of records. This is in order to prevent threats made possible by the aggregation of data. All additions to a patient's record must be attributable. Essentially, all clinicians may add anything they wish to the record of a patient in their care, but none shall have the ability to destroy any of the record. The security policy proposed by Anderson is as follows:

- each patient record must name the clinical professionals who may have access to all or part of the record,
- patients must be notified of the names on their records and of all additions to it,
- the patient must consent to all clinicians named on his record,
- only clinical professionals may be added to a name on a record,
- special notification must be given where there is a threat due to aggregation of data,
- all clinicians who append to a record must have the addition marked with his name and the date and time,
- no clinician can modify or delete any part (or all) of a patient's record.

Anderson [1] proposes a solution which uses an ACL in each patient's record. This ACL names the clinicians and the access they are allowed to the public or confidential part of the record.

3.3 Discussion

As can be seen from these two examples, policies specify actions at different levels. There are those aspects which may be directly interpreted as constraint rules e.g. 'no users other than those involved should see the state of an object' and 'each patient record must name the clinicians who may have access to it'. These constraints, although sometimes expressed as a negative rule, can also be interpreted as a positive authorisation rule i.e. an authorised role holder may see the state of an object' and 'clinicians named in a record may view a patient's record'. Such rules may be enforced using an ACL and can be seen as an authorisation policy [10].

Using an ACL to enforce a policy, however, does not encapsulate all the policy requirements. For example, the right to delegate and the different levels of delegation allowed can not easily be represented in an ACL. Also the authority that grants access (authorises that a name is added to an ACL) is not explicitly represented. Consider the patient's need to consent to a clinician being included on his record. Once a name is in an ACL access is granted, yet there is no provision made to indicate (other than procedural) that a patient has actually consented to this inclusion.

There are other aspects of the policies however, which are obligations e.g. 'an object must be finalised by a role holder', and 'special notification must be given where there is a threat due to aggregation of data'. These are also rules, but these cannot be expressed using conventional access control mechanisms. These need to be expressed in such a way that fulfilment of the obligation can be enforced when the action takes place.

A policy divides into the two 'types' of rules, namely those that can be applied by checking the access control mechanism and the ones that are more abstract and are not so easy to express using straight forward access control mechanisms. We now show how we can move one step further towards expressing some of the more abstract requirements.

4. The SAProxy Scheme

Let us consider the SAProxy authorisation scheme [7, 8] as an alternative to using ACLs to enforce a policy. The SAProxy scheme is a mechanism, based on tokens of delegation and public key encryption [12]. An SAProxy is an extended capability naming the principal to whom it is issued and digitally signed [5] by the issuer of the SAProxy. 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.

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.

The authority, expressed in SAProxy tokens is unforgeable as they are digitally signed and can be produced and verified locally by each principal. Only the principal to whom the token is issued can use it, so there is no problem with propagation. These features make SAProxies particularly suitable for a distributed open environment. The following is a brief description of the mechanism. For a full description see [9].

4.1 The SAProxy Mechanism

SAProxies are based on public key encryption, so all principals involved must have a public key pair. Public keys are certified and signed by certification authorities.

A principal can then be identified and referred to by his public key certificate (PKC), because these are unique, and the PKC itself can be referred to by its signature block, e.g. a user A can be referred to by his PKC signature, $sigC_A$ [3].

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 forms part of the token. SAProxies are referred to by their signature block.

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^-

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 and this can be done in a domain that is remote from the one in which the SAProxy was generated.

Within a single SAProxy, the information shown above need not be explicitly stated but may be referred to by other SAProxies. If B 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* $sigD_{C:B}$

The level of delegation can be specified by having a SAProxy constraint in the access rights field which indicates how many times a SAProxy can be delegated. Thus these tokens of delegation can be bound together to show the grounds upon which access to an object is

granted. It is important to note that because these SAProxies are signed by each issuer, they can be verified by any party, with the appropriate PKC, and not just by the object server.

The SAProxy scheme allows for the definition of roles. A role can be named and authority granted to that role instead of a principal. Then a principal can be granted authority to act in a role instead of being granted access control directly.

A principal G, with authority to assign roles to other principals, can issue a SAProxy as follows:

$R_{B:G}$: $sigC_B$ $sigC_G$ *null* W $sigR_{B:G}$

In $R_{B:G}$, G authorises principal B to act in role W. The token is signed by G.

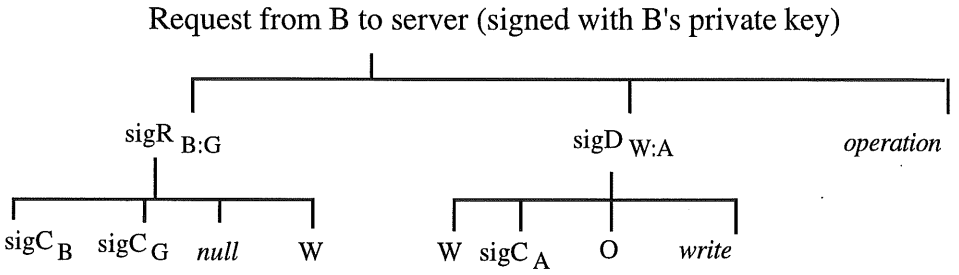
Delegation can be applied not only to principals, but also to roles although these need to be bound to or invoked by a principal before the authority delegated can be exercised.

Now A can grant access to the role W instead of directly to B.

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

To write to object O, B then has to send all his tokens of authorisation to the object server with his request to operate on O. B has to send his role SAProxy from G, $R_{B:G}$, and the authorisation SAProxy for W from A, $D_{W:A}$, together with the operation he wants to perform. He binds these together by signing them with his private key to show that he is the true owner of the tokens.

The object server checks that the role, W, in $D_{W:A}$ is the same role, W, granted to B in $R_{B:G}$. The line of authority can be seen in the following tree:



The only operation that W is authorised to request is write.

The role, W, may be assigned to different principals without having to redefine W's authority. The SAProxy scheme allows the following; 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 set in an object's ACL and finally, a consistent approach is used at different levels of authorisation which makes the enforcement and auditability of this authorisation scheme easy to implement and verify.

5. A Possible Way Forward

Authorisation policies can be expressed and enforced using ACLs. However, ACLs themselves must be managed securely and this is more naturally done centrally. It must not be possible for anyone, except the 'ACL manager' to modify an ACL. When employing RBAC there is also the added requirement to verify that a principal is authorised to act in a role. SAProxies provide a more flexible means of expressing the authority that has been granted to a principal and the authority under which an action is being performed, whether in a centralised or distributed system. We now illustrate how the two example policies may be represented using SAProxies.

5.1 The Exam Paper

Consider the role the examiner that sets the paper, EX1, has to play. Each party has a public key pair and a PKC. Each PKC is referred to by its signature block e.g. the chair's PKC is referred to as $sigC_H$. The chair of the board, H, assigns the role of EX1 to Jones, J.

$R_{J:H}$: $sigC_J$ $sigC_H$ *null* EX1 $sigR_{J:H}$

The role EX1 is then granted the following authority to create and write the exam paper:

$D_{EX1:H}$: EX1 $sigC_H$ paper *create/read/write* $sigD_{EX1:H}$

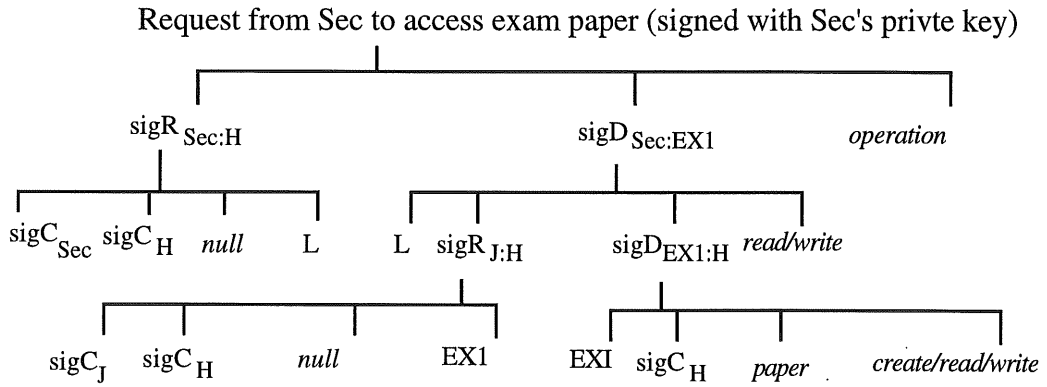
To enable EX1 to get clerical help the clerical staff must be taken from a specified pool. The chair, H, can specify the pool by assigning a role, L, to all members of the pool.

$R_{Sec:H}$: $sigC_{Sec}$ $sigC_H$ *null* L $sigR_{Sec:H}$

Then Jones can, in his role as EX1, delegate some of his authority to L to help him with his work.

$D_{Sec:EX1}$: L $sigR_{J:H}$ $sigD_{EX1:H}$ *read/write* $sigD_{Sec:EX1}$

The flow of authority is shown in the following tree.



It must be noted from the above that the only two operations that Sec is authorised to request are read and write.

5.2 The Medical Information System

To illustrate the use of SAProxies in this example we shall just look at the authority required to access the confidential part of a patient's record. Each patient's record has an ACL which holds the names of the clinicians with access to all or part of that record. The patients have to consent to each clinician that is authorised. Instead of using an ACL each clinician is issued with SAProxies to give him the necessary authority.

One of the requirements of the policy is that only clinical professionals may be added to a record. A clinician (Dr. X) is authorised to practice by his health authority or hospital . The hospital authorities (HA) therefore gives Dr X a role SAProxy which authorises him to act in the role of clinician professional, CP, in a particular department or surgery.

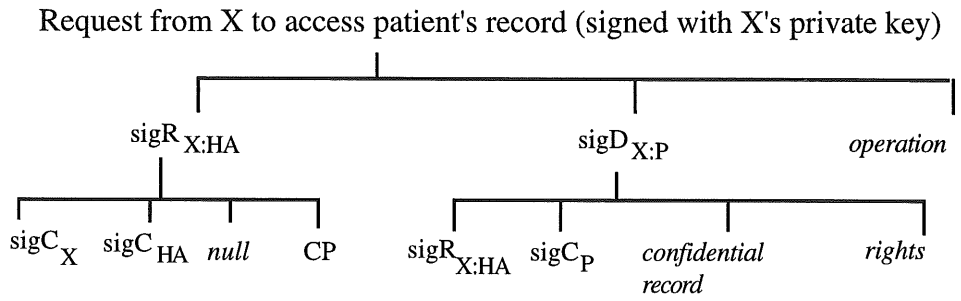
$R_{X:HA}$: $sigC_X$ $sigC_{HA}$ *null* CP $sigR_{X:HA}$

Another policy requirement is that patients must consent to all clinicians with access to all or part of their records. Each patient (P) will have a public key pair and a PKC ($sigC_P$) recognised by the hospital. Each patient then gives the appropriate clinician authority to access the confidential part of his record in a SAProxy. In contrast to the first example, patients do not consent to just anyone assigned to the role of a professional clinician having access to the confidential part of their records, but to specific clinicians. Therefore patients must grant

authority to named clinicians, in their role as a CP and not to the CP role. This is satisfied in the following SAProxy.

$D_{X:P}$: $sigR_{X:HA}$ $sigC_P$ *confidential record* *rights* $sigD_{X:P}$

The clinician's authority to request access to the record can be seen below



The operation requested by X must be within the rights he has been granted in the SAProxy. The difference between this and the first example is that authority to access the confidential part of the record is granted only to a particular clinician and not to the role of CP. Authority to access the rest of the record (generally available to all clinicians in the department or surgery) can be given to the role CP.

5.3 Discussion

These examples show that SAProxies are a powerful alternative to ACLs. SAProxies embody the concept of delegation of authority within the problem of access control. By using a scheme that works on this principle it is possible to represent more aspects of a policy more clearly.

- It is easy to see who has granted authority to a principal as the authorising principal is explicitly stated (this is only implied in an ACL i.e. the patient giving his consent).
- It is easy to see in what capacity a principal has done a particular action, as the role being played is also explicitly stated i.e. Jones gives access to Sec in his capacity as EX1. (Again with ACLs the authorising principal is implied and in the above situation, Jones would need to be able to modify the security template when he gives authority to Sec).
- It is easy to see, and verify, from the tree of tokens whether the authorisation policy can be enforced. The line of authority can be followed i.e. the tree can be traced to verify that the origin of authority complies with the policy. In the examples discussed the verifier has only to know the chair's and the hospital authority's public keys so as to recognise their signatures. From then on all other authority can be verified.
- SAProxies are easily employed across different management domains. A role can be defined in one domain and a principal in another domain can be authorised to act in it. All the authorisation tokens can be generated locally and verified in any domain without any fear of forgery or of tampering with genuine tokens. Furthermore, principals with authority are able to generate the authorisation tokens themselves and do not have to rely on (and trust not to abuse their powers) any system official to modify the security components of their system e.g. to alter an ACL.

One of the arguments for ACLs used by Anderson in [1] is that ACLs are more practical because patients have only one doctor and doctors have many patients and would therefore need to hold and manage many capabilities. However, because SAProxies are unforgeable, tamper-proof and attributable, there is no reason why each clinician's SAProxy to access a particular record cannot be held in the record.

Although we have shown how an authorisation policy can be enforced using SAProxies, we have not discussed any aspects of the obligation policy. However if we analyse what has been achieved in the examples it can be seen that some of the obligation policy (but not all) has also

been expressed by the SAProxies e.g. only clerical staff assigned the appropriate role by the chairman can work on the exam paper, and the fact that patients grant authority to their clinicians to add to their records satisfies the need for them to be informed of all additions and explicitly shows that their consent has been given.

SAProxies provide a notation, the SAProxy trees, with which to express some of the policy rules. It is very easy for a policy checker to traverse this tree and verify that the policy is satisfied, and then the policy can be enforced by confining all actions to only those authorised. Each tree, however, is only a specific instance of the fulfilment of the policy which can easily be checked. The following issues still remain. It is not possible, with the tools we have, to express the full generality of a policy. SAProxies can be used to enforce a policy, but can they be used to describe a policy? Is it possible to have a policy checker that is generic so that given a 'description' of the SAProxies expected it can check that it gets the right ones?

Obviously, a policy checker can be designed to check these requirements explicitly, but then we are making the policy part of the implementation of the application. Do we then define policy merely as who can invoke the functions that are implemented? We suggest that policy should not be a part of software implementation, but independent of it.

6. Conclusion

The discussion presented in this paper suggests the following: that a security policy can be viewed as two distinct policies; the authorisation policy which defines actions allowed and the obligation policy which defines responsibilities that need to be fulfilled. There are therefore two types of policy 'rules'. The authorisation rules can be translated easily into an access control mechanism., but the obligation rules can not. We have shown that a security policy implemented in terms of just an access control list does not satisfy all the requirements of the policy because there are parts of the policy that cannot be expressed in those terms. We have shown that using this technique some policy requirements are implied rather than explicitly stated and verifiable. Furthermore, ACLs are cumbersome when operating over a distributed environment in that they can only be modified and checked locally .

We then proposed the use of SAProxies to express the security policy and have shown that when using this technique all authorisations are explicitly stated and verifiable. Authority can be granted locally by the relevant party without the involvement of any system management and this authority can be checked remotely without the need for any on-line communications. We have also shown that some of the obligation policy requirements are also clearly stated when using SAProxies.

Although we have shown that it is easy, using SAProxies, for a policy checker to verify that a role holder is properly authorised, we have no means, as yet, of stating explicitly other requirements such as that the role holder must be authorised by a particular principal or that the role holder must be from a specific pool.

We also have not provided a means of fully expressing an obligation policy i.e. we have not sufficiently developed a notation with which to state the duties required of a particular role against which to check any actions. A policy checker can verify that an object has been finalised (signed) by a role holder, but again we have not explicitly been able to express the need for this signature. We have described a way of ensuring that all constraints can be clearly stated and enforced but not what the duties should be. There is, therefore, a need for the SAProxy notation to be developed further in order to satisfy the above requirements.

Acknowledgements: My thanks to John Bull from ANSA, Architecture Projects Management Ltd. in Cambridge for his ideas on this subject.

References

[1] Anderson R.J. A Security Policy Model for Medical Information Systems *1st Conference - Computer and Communications. Security '93* ACM VA, USA. November 1993

[2] Boswell A., Specification and Validation of a Security Policy Model *IEEE Transaction on Software Engineering* 21(2):63-68 February 1995

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

[4] Coulouris G., Dollimore J. *Protection of Shared Objects for Cooperative Work* TR 703 Department of Computer Science, Queen Mary and Westfield College, University of London. September 1995.

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

[6] Lewis L. Implementing Policy in Enterprise Networks *IEEE Communications Magazine* pp.50-55 January 1996

[7] Low M.R., Christianson B. A Technique for authentication, access control and resource management in open distributed systems *IEE Electronics Letters* 30(2):124-125 January 1994

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

[9] 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.

[10] Lupu E.C., Marriott D.A., Sloman M.S., Yialelis N. A Policy Based Role Framework for Access Control *First ACM/NIST Role Based Access Control Workshop, Gaithersburg, USA.* ACM December 1995

[11] McLean J. Reasoning About Security Models *Proceedings of the IEEE Symposium on Security and Privacy* IEEE pp.123-131 April 1987

[12] Rivest R.L., Shamir A., Adleman L. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120-126, February 1978.