Expressing Security Policies for Distributed Objects Applications

Popescu, B.C.; Crispo, B.; Tanenbaum, A.S.

published in
Proc. 11th Int'l Workshop on Security Protocols
2003

document version
Publisher's PDF, also known as Version of record

Expressing security policies for distributed objects applications

Bogdan C. Popescu, Bruno Crispo, Andrew S. Tanenbaum, Maas Zeeman
Vrije Universiteit, Amsterdam
{bpopescu,crispo,ast,mmzeeman}@cs.vu.nl

February 5, 2004

1 Introduction

In this paper we describe the design and implementation of a policy engine for enforcing security policies for distributed object applications. We show how our design can be integrated as part of the Globe [11] system - a middleware for supporting wide-area replicated objects.

While extensive work has been done in the area of security policy languages and policy engines, this paper makes two important contributions: first we identify a number of security policy requirements that arise in the context of replicated applications, more specifically, the need for policy mechanisms to express different amounts of trust one wants to place into different replicas of the same service. Second, we come up with a design that bridges the gap between an abstract security policy description and the actual service implementation. This is consistent to our goal to provide a policy engine at the **middleware level** which would make it simpler for application developers to integrate the policy engine with their applications. Traditional policy engines [2] work at a more abstract level, which in theory makes them very versatile, but in practice means that developers need to write rather complex translators (for passing parameters and environment variables) in order to bridge the gap between the engine and the application.

The rest of the paper is organized as follows: in Section 2 we give an overview of the Globe system, which is the testbed for the policy engine we have developed. In Section 3 we describe the trust model for Globe applications; our policy language is specifically designed to support this trust model. In the next three sections we describe the policy language constructs, grouped into constructs for supporting administrative policies, access control and method execution policies. Finally, in Section 7 we give an overview of our implementation, in Section 8 we examine related work, and in Section 9 we conclude.

2 An overview of Globe

Globe is a distributed system based on replicated shared objects. While the idea of encapsulating functionality into objects is not new (systems like Corba [1], Legion [5] or DCOM [4] rely on this paradigm), what makes Globe unique is that objects not only can be used by a large numbers of users on different machines through remote procedure calls, but also can be physically replicated on many hosts at the same time to improve performance.

The central construct in the Globe architecture is the distributed shared object (DSO). As shown in Figure 1 a DSO is built from a number of **replicas** that reside in a single address space and communicate with replicas in other address spaces. All the replicas that are part of a DSO work together to implement the functionality of that DSO. A replica consists of the code for the application (the code that implements the functionality of the DSO that replica is part of), the part of the DSO state the replica stores, and the replication mechanism. A replica can be hosted by any Globe-aware server connected to the Internet. To be Globe-aware, a server needs to run a special daemon program - the **Globe object server** that provides facilities for hosting, remote creation and destruction of replicas. We also provide a special **Globe Location Service** [10] where DSO register the contact points for their replicas, so that their clients can easily find them.

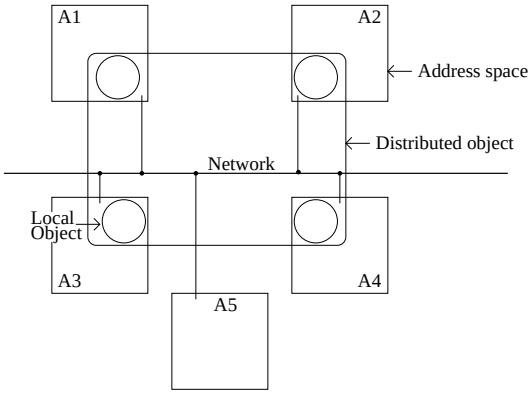

Figure 1: A Globe DSO replicated across four address spaces

In the context of a Globe, a **client** is an entity that uses a DSO by invoking one of its public methods. Clients and replicas are assumed to operate in different address spaces, thus, in order to use a DSO, the client first has to find a replica part of that DSO, connect to it and then send a remote method invocation request. At first, it may seem strange to also consider the clients of an application (modeled as a DSO) as part of that application. However, as we will see in Section 6, here this makes sense because in Globe the clients of a DSO are also responsible with enforcing (part) of its security policy.

3 Trust Model

The cornerstone of the Globe trust model is that individual DSOs are fully in charge with their security policies. This means a Globe object **does not need** any external trust broker in order to run securely. Because DSOs can be massively replicated across wide-area networks, we have chosen public key cryptography as the basic cryptographic building block for implementing the DSO trust hierarchy. The alternative, namely to use only shared secret keys, has the disadvantage that we need to take special measures to reduce the number of keys, for example, by using a Key Distribution Center. Although public keys introduce their own scalability problems, such as those related to certificate revocation, we have nevertheless decided to associate public/private key pairs with all distinct Globe entities (DSOs, replicas, clients), believing that these are more easy to deploy in a large-scale system.

We require that each DSO has a public/private key pair, which we term as the **object key**. The object key acts as the ultimate source of trust for the object, and any principal that has knowledge of the object's private key can set the security policy for that object (we term such a principal the **object owner**). For a given DSO it is assumed that the object key is known by all entities (clients and replicas) part of that DSO. This can be accomplished through the use of a PKI or through some other out-of-band means.

We also associate a public/private key pair with every DSO replica and client (we call this the **replica key** and the **client key**). Permissions are then associated with these public keys. For example, in the case of a client, we associate the client's public key with the methods that client is allowed to invoke. For replicas, an example of permission that can be associated with its public key, is which of the DSO's methods the replica is allowed to handle.

For a given DSO, the set of all permissions associated with all its clients and replicas form the **security policy** for that DSO. Whenever two entities part of the same DSO interact, they first need to authenticate each other, and then check the DSO's security policy to ensure their interaction follows that policy. For example, when a client invokes a method on a replica, the replica needs to check the security policy to ensure the client is allowed to invoke the method, and the client needs to check the policy to ensure the replica is allowed to handle it.

There are many ways this security policy can be expressed; for example, one can envision a (very large) table storing the public keys of all clients and replicas and listing the permissions associated with each of them. However, such a solution clearly does not scale, since this table would have to be distributed to all entities part of the DSO. Furthermore, updating such a highly replicated table would be a daunting task.

The solution we have envisioned for Globe is inspired by the work done on Role Based Access Control [8]. The idea is to group entities (clients, replicas) with equivalent security properties into **client/replica roles**, and express a DSO's security policy based on these roles. The assumption is that the number of roles is not very large and fairly static (for a given application, one does

not have to add a new class of clients every day); We also assume that the set of permissions associated with a role does not change frequently. Based on these assumptions, a security policy expressed in terms of such roles has two big advantages: it is quite compact (because the number of roles is much smaller than the number of entities that are mapped in these roles), and does not change frequently. Thus, it is scalable to distribute this security policy to all entities part of a DSO.

Besides this role-based policy description, we also need to provide the mapping of clients/replicas (identified through their public keys) to the roles they have been assigned. This is accomplished through **role certificates** - digital certificates that bind an entity public key to the role that entity has been assigned as part of the DSO's security policy. Such role certificates are issued by DSO entities (clients or replicas) that have been assigned administrative privileges. For a given DSO, we define an **administrative role** as the set of all administrative entities with equivalent security properties, which in this case means that entities in the same administrative role are allowed to issue the same types of role certificates. Each DSO is allowed to define its own administrative hierarchy; however, the DSO's owner will always be the root of such a hierarchy.

Once we introduce the client, replica and administrative roles, we can conceptually divide a DSO policy expressed in terms of these roles in three parts: the access control policy is mostly concerned with expressing permissions associated with client roles; the invocation policy is mostly concerned with expressing permissions associated with replica roles. Finally, administrative policies deal with describing how administrative roles are allowed to assign other roles. In the following sections we will look at each of these policies in detail.

4 Expressing Administrative Policies

As we explained in the previous section, role certificates are issued by clients or replicas that have been given administrative privileges as part of a given DSO. Each DSO has at least one administrative entity - the object owner - which by default has all the possible administrative privileges (it is allowed to delegate every possible role in the DSO's role hierarchy).

At first, it may seem strange to have both clients and replicas as administrators for an object, since one would usually associate a human with such a role. Administrative replicas come in handy when we deal with massively replicated DSOs. For such DSOs, a highly dynamic pattern in client requests can be better handled by creating new replicas on the fly, in places where most of the client requests come from. In such a scenario, one client administrator can start a number of administrative replicas, and issue administrative certificates granting them the right to issue replica certificates. These administrative replicas could in turn monitor client requests and create regular replicas in places where they can better handle these requests.

The only type of privileges that can be associated to administrative roles is the right to delegate other roles. Thus, an administrative role can be fully

described by listing all the roles it can delegate. An intuitive way to see a DSO's role hierarchy is as a directed graph, with each role corresponding to a node; in such a graph, an edge from node A to node B, implies that role A is an administrative role, and it has the right to delegate role B under the DSO's security policy. Based on what we discussed so far, such a graph needs to have the following properties:

- it has exactly one node of in-degree 0; this is the object owner role, which is implicitly assigned to the principal that has access to the DSO's private key.
- all nodes corresponding to administrative roles (except for the object owner role) have non-zero in-degrees and out-degrees (because an administrative role should be able to delegate at least one role).
- all nodes corresponding to client or replica roles must have a zero out-degree (because they should not be able to delegate any roles).

In addition to this, we would like our role hierarchy to be monotonic, namely a role with less privileges should never be able to issue a role with more privileges. This can be accomplished by enforcing the following extra rules on the role graph:

- the graph should not have any cycles of length greater than one edge. By allowing one-edge cycles we allow administrative roles to replicate themselves.
- the set of zero out-degree children of any node is a subset of the set of zero out-degree children of its parent. This ensures that no administrative entity can get more privileges in issuing client/replica roles by delegating itself to a lower administrative role.

Designing a role graph with the above properties is in fact equivalent to describing the DSO's administrative policy. For Globe DSOs this is done by the object owner who needs to describe the graph in a policy language. One way of un-ambiguously describing a directed graph is by describing all its edges; this can be done using a language construct of the type:

$Role_A$ **canDelegate** $Role_B$

Here, $Role_A$ and $Role_B$ are role names. A DSO's administrative policy is then fully described by a set of such statements. A policy interpreter can then process all these statements and construct the role graph. The graph is first checked to follow the monotonicity properties. After that, all nodes with non-zero in- and out-degrees are interpreted as administrative roles. All the other ones (except for the object owner role) are interpreted as either client or replica roles. How we distinguish between these two, we will describe in the following sections. Once the DSO's administrative policy has been designed and checked for monotonicity, it needs be securely distributed to all DSO's entities. This policy is then

used to verify the role certificate chains that DSO entities exchange when they authenticate each other. A certificate chain is considered valid if the signature chaining is correct and the role delegation described by the chain follows a path in the role graph. Thus, each DSO entity needs to be provided with a policy engine, which, after initialized with the DSO's administrative policy can answer queries of the type:

isValidChain(*certChain***)**.

where *certChain* is a chain of digital certificates corresponding to a sequence of DSO entities (clients and replicas) delegating DSO roles to each other. The **isValidChain()** query returns $True$ if the sequence of roles in *certChain* corresponds to a valid root path in the DSO's role graph (this means the chain starts with a certificate signed with the DSO's key, and for each certificate, the role in the certificate **canDelegate** the role in the next certificate in the chain) and $False$ otherwise.

5 Expressing Access Control Policies

Besides administrative privileges, a DSO entity can also have method invocation privileges; such privileges describe which of the DSO's methods that entity is allowed to invoke. Method invocation privileges are not restricted only to the DSO's clients, replicas can also invoke methods on each other; a good example of this is replicas of a DSO that implements a master/slave replication strategy: whenever the DSO's state changes, the master replica invokes a special method $Invalidate()$ on all slave replicas. In order to get the new state, the slaves then invoke $StateUpdate()$ on the master. Thus, the master replica role needs to have invocation privileges on the $Invalidate()$ method, while the slave replica role needs invocation privileges on the $StateUpdate()$ method. However, the slave replica role should not be allowed to invoke $Invalidate()$, otherwise, a malicious slave would be able to propagate state changes (it is assumed the master is more trustworthy in this case).

A method invocation privilege can be expressed through a policy statement as the following:

Role **canInvoke** $Method$ **underConditions** *Conditions*

here, *Role* is a non-administrative role previously declared in a **canDelegate** statement; $Method$ is the name of one of the DSO's methods; *Conditions* is a boolean expression that puts constraints on the way the $Method$ can be invoked by *Role*; the terms that can appear in *Conditions* are the following:

- the parameters passed to this method; in this way, *Conditions* can impose certain parameter ranges for *Role* invoking $Method$.
- external functions; these have to be separately defined so the policy engine knows how to invoke them; such external functions can impose constraints

on the way *Role* can invoke $Method$ based on things like the object's state, the resources available on the system running the replica, time of the day, or the location where the request originates. The only requirement here is that such external functions are synchronous - this ensures the policy engine cannot be blocked on an external function.

- additional attributes (expressed as name-value pairs and always interpreted as strings) present either in the certificate chain associated with the caller, or in other certificates provided by the caller.

A DSO's access control policy can then be fully described through a set of such **canInvoke** statements. Again it is the object owner that designs this access control policy and is responsible with securely passing it to all DSO replicas (only replicas need to do access control checks, since clients can only invoke methods, they cannot execute them). Each DSO replica then stores this policy in its policy engine, and whenever receives a request from an (already authenticated) client, it checks the request against the policy through the following query:

isAllowed(*Role*, $Method$, $Parameters$**)**

where *Role* is the role the client has authenticated with, $Method$ is the method invoked by the client and $Parameters$ represent the actual parameters. The **isAllowed()** query then returns either $True$ if there is a **canInvoke** statement that allows the invocation of $Method$ with the given parameters by *Role*, considering all external conditions that may apply, or $False$ otherwise.

6 Expressing Method Execution Policies

The last type of privileges that can be associated with a DSO entity are method execution privileges. The need for expressing such types of privileges is a direct consequence of the fact that Globe DSOs can be replicated over hosts of various degree of trustworthiness. In such a setting, it makes sense for the owner of a DSO to restrict the execution of the most security sensitive operations to the replicas that run on the most trustworthy servers (at least from from the owner's perspective). The owner does this by indicating to clients which replicas can be contacted for each type of method invocation. This type of "reverse access control" can also be seen as a way to label quality of service and/or trust level of replicas to clients.

While the task of enforcing the access control policy relies with the replicas, it is clients that enforce this method execution policy. It is assumed that a client that has been allowed to invoke a method under the DSO's access control policy has all the interest to ensure that invocation is sent to a replica that is allowed to execute it under the DSO's method execution policy.

A method execution privilege can be expressed through a policy statement as the following:

RoleExpr **canExecute** *Method* **underConditions** *Conditions*

Here, *RoleExpr* is an expression of the form:

```
<RoleExpr>:: <PrimaryTarget> | <RoleExpr> "&&" <SecondaryTarget>;
<PrimaryTarget>:: <RoleGroup>;
<RoleGroup>:: <Role> || "Traceable(" <Role> ")" ||
      <PositiveInteger> "*" <Role> ||
      <PositiveInteger> "* Traceable(" <Name> ")";
<SecondaryTarget>:: <RoleGroup> ||
    <PositiveInteger> "\%" <Role>;
```

Such an expression is used to describes a group of replicas in possibly different roles, and the way these replicas need to be contacted by a client that wants to invoke *Method* on the DSO. The reason why more than one replica may need to be contacted when invoking a method is Byzantine fault tolerance; here we envision three basic techniques that can be used to achieve such Byzantine fault tolerance for a DSO consisting of many replicas of various degrees of trustworthiness:

- **replicated invocation**: the same method is invoked on a number of (less trusted) replicas. The result is accepted only when a certain number of them agree on the return value. Thus, in order to make a client accept an incorrect result, a number of malicious replicas would have to collude.
- **traceable results**: a (less trusted) replica executing a method has to sign (with its private key) the invocation request and the return value. The client can then forward this traceable request-result pair to a more trustworthy replica that may audit the result (by executing the method again and comparing the result). Thus, less trusted replicas acting maliciously can be traced and eventually excluded from the DSO.
- **double-checking**: the client first invokes the method on a less trusted replica, and then may double-check the result with a more trustworthy replica. However, in order to avoid overloading the trusted replica, the double-checking is done statistically (for each request there's only a small probability that request will be double-checked).

For example, the expression $3 * Role_1 + 2 * Role_2$ specifies that 3 replicas in role $Role_1$ and 2 replicas in role $Role_2$ have to agree on the result of invoking $Method$ before the client accepts the result. As another example, the expression $Traceable(Role_1) + 5\%Role_2$ specifies that $Method$ can be invoked on a replica in role $Role_1$, which has to sign the result; furthermore, the result should be double-checked with a replica in role $Role_2$ with a probability of 0.05.

It is important to understand that such mechanisms for achieving Byzantine fault tolerance work only with methods that are idempotent (invoking them multiple times has the same effect as invoking them once). It is the DSO owner's

responsibility to ensure that only such idempotent methods are marked as "multiple invocation" in the DSO security policy file.

The other elements in the **canExecute** statement have the same meaning as in the **canInvoke** statement: *Method* is the name of one of the DSO's methods while *Conditions* is a boolean expression that puts constraints on the way the *Method* can be executed by *RoleExpr*; again, conditions can be placed on parameter values (only certain ranges allowed), possibly time of invocation.

A DSO's method execution policy can then be fully described through a set of such **canExecute** statements. Again, it is the object owner that designs this policy and is responsible with securely passing it to all DSO clients and replicas (remember from the previous section that both clients and replicas can invoke the DSO's methods). Each DSO entity then stores this policy in its policy engine, and consults it before issuing a method invocation request by making the following query:

whoCanDoIt(*Method*, *Parameters***)**

where *Method* is the method to be invoked and *Parameters* represent the actual parameters. The policy engine then searches through all the **canExecute** statements in the policy and stops whenever it finds one that matches *Method*, and for which the *Parameters* value satisfy the **withConditions** part of the statement. It then returns the *RoleExpr* from the selected statement. The entity that does the invocation then needs to locate replicas in the roles described in *RoleExpr* and send the method invocation request to each of them.

7 Implementation Overview

We have implemented a policy engine based on the design outlined in the previous sections; the idea is to integrate this engine with the Globe middleware.

The policy engine has two separate parts. One part deals with certificate chain validation, the other part with the security checks needed for DSO method invocation.

The certificate chain validation engine has a generic core which is based on the semantic model of trust management engines (TME) defined in [12]. This generic core is an implementation of the proof-check method. The core can be used for any trust management system which can be expressed with the semantic model. This approach allows easy experimentation with new kinds of certificate languages, and new ways to delegate authorizations to principals.

An overall policy constructed by the DSO owner defines the role-hierarchy, and permissions. The implemented trust management system uses the role-hierarchy to check that principals only delegate lower or the same role to other principals.

The other part of the policy engine is responsible with interpreting the **isAllowed** and **canExecute** statements related to method invocation and method execution.

Permissions contain a condition expressions, which allows the DSO owner to express restrictions on the actual parameters passed to a DSO. The syntax of the condition language is based on the syntax of Java expressions. To be usable for sound access restrictions the expression language uses a very strict static type checking algorithm, which makes sure that the evaluation of the conditions can never get stuck due to illicitly typed expressions.

The expression language has 6 base types, like: *int, long, float, double, char, boolean*, and *string*. When conditions are placed on DSO method parameters declared as complex Java classes, the class-name needs to be declared as a new foreign type and the beginning of the policy file. These foreign types can only be compared using the $=$, and $\neq$ operators. When the foreign class also implements the Jave *Comparable* interface the $<$, $>$, $\geq$, and $\leq$ operators can be used directly. To give the policy writer even more control over values of foreign types it is also possible to define regular Java functions operating on such values. The strict static type checking algorithm makes sure that these function can only be called with parameters of exactly the same types as used in the definitions. This makes the invocation of the methods safe, because it does not allow one to pass parameters constructed from sub-classes not known at the time when the policy was written.

8 Related Work

In this section we compare our solution with a number of other trust/certificate management systems: KeyNote [2], RTML [7], XACL [6], X.509 [9] and PGP [3].

The KeyNote system [2] provides a simple language for describing and implementing security policies, and digitally signed credentials. KeyNote unifies the notion of a security policy with that of a credential. This idea is powerful, and allows for sophisticated setups. The drawback of this solution is that it is not possible to just look at the policy to audit the security setup.

KeyNote is more general than the policy language we propose, so it should be possible to express any Globe policy in KeyNote. However, our policy language makes it very easy to express policy constructs related to byzantine fault tolerance and replicated invocation and auditing, which become more complicated with Keynote. Another difference is that we allow tight integration between the policy engine and the secured application. Our policy language also has the possibility to define and call foreign methods from the application. This allows an application programmer to make policy decisions based on application specific values, something which is not possible with KeyNote.

The Role-based Trust-management Markup Language (RTML [7]) has much in common with our solution. It is also role based, and is embedded in a programming language. Just like our solution, the credentials and the policies are encoded in XML. The main difference between our solution and RTML is that we do not allow arbitrary language constructions in credentials. Our credentials can only contain constructs which deal with the delegation of roles,

making them simpler. RTML also does not support a tight integration between the security system, and the application.

The XML Access Control Language (XACL see [6]) has a completely different goal as our system. XACL was designed to create access policies for XML documents, while our system was designed to create policies for a distributed object system.

Pretty Good Privacy (PGP [3]) and X.509 [9] are well known certificate management systems. Both of them focus entirely on authentication, while our solution focuses on authorization. X.509 certificates bind so called Distinguished Names (DN) to public keys, while PGP certificates bind e-mail addresses to public keys. On the other hand, for the solution we propose, we bind roles to public keys associated with DSO entities. The main difference in this approach is that all possible roles are known in advance. This is not the case for DNs, and e-mail addresses. Both systems focus only on the authentication problem, and leave the authorization problem entirely to the application programmer.

9 Conclusion

In this paper we have described the design and implementation of a policy engine for enforcing security policies for distributed object applications. In our design, we explicitly take into account object replication, which introduces specific requirements, such as the need for policy mechanisms to express different levels of trust one wants to put on different object replicas. Another distinctive feature of our approach is the aim to provide our policy engine at the middleware level; as a result, our policy language has features that bring it closer to an actual programming language (Java), which should make easier to integrate it with applications.

References

[1] The Common Object Request Broker: Architecture and Specification. www.omg.org, Oct 2000. Document Formal.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust-Management System, Version 2. RFC 2704, September 1999.

[3] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC 2440, November 1998.

[4] G. Eddon and H. Eddon. *Inside Distibuted COM.* Microsoft Press, Redmond, WA, 1998.

[5] A. Grimsaw and W. Wulf. Legion - a view from 50000 feet. In *Fifth IEEE Int'l Symp. on High Performance Distr. Computing.* IEEE Computer Society Press, Aug 1996.

[6] M. Kudo and S. Hada. XML Document Security based on Provisional Authorization. In *Proc. 7th ACM Conf. on Comp. and Comm. Security*, November 2000.

[7] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland*, May 2002.

[8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–48, Febr. 1996.

[9] I. T. Union. Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks, March 2000.

[10] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Mag.*, pages 104–109, January 1998.

[11] M. van Steen, P. Homburg, and A. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78, January-March 1999.

[12] S. Weeks. Understanding Trust Management Systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 94–105, May 2001.