A Conceptual Authorization Model for Web Services[1]

Paul J. Leach, Chris Kaler, Blair Dillaway, Praerit Garg,
Brian LaMacchia, Butler Lampson, John Manferdelli, Rick Rashid,
John Shewchuk, Dan Simon, Richard Ward
Microsoft Corporation, Redmond, Washington, USA

This paper describes a conceptual authorization model for Web Services. It is an adaptation of those of Taos [Lamp92] and SDSI [Lamp96] with terms changed to correspond more closely to those introduced with the WS-Security model [WS02]. In contrast to the more formal and mathematical presentation used for Taos and SDSI, this presentation is conceptual and informal, which hopefully may provide more intuition for some readers; it also might provide an outline for the class hierarchy of an object-oriented implementation.

In addition, this model abstracts away from issues of distribution and network security such as authentication [Need78] and encryption (for example, by assuming that messages include the unforgeable identity of the sender and are private and tamperproof) so as to focus on authorization, but it does deal with the extensibility and composability of security services, and partial trust. It also abstracts away from issues of syntax and encoding (for example, ASN.1, proprietary binary formats, and XML) and focuses on semantics.

The following figure illustrates many of the elements of this model that will be described in this paper:

[1] This paper was written for a symposium in honor of Roger Needham, February 2003, and published in *Computer Systems: Theory, Technology, and Applications*, K. Sparck-Jones and A. Herbert (editors), Springer, 2004, pp 137-146.

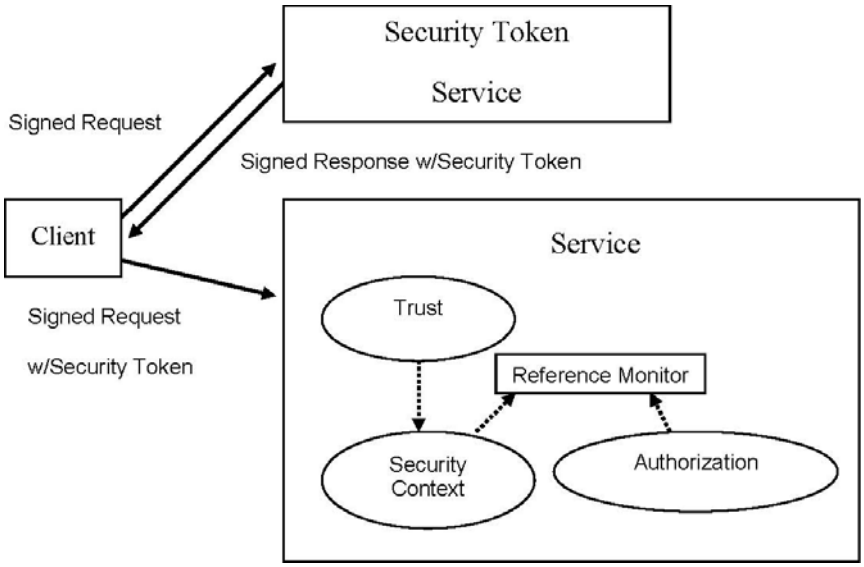

Figure 1.

Basic computational model

Computations are done by running *programs* in *processes* which contain one or more parallel *threads* of execution. Processes have separate address spaces and are isolated from unwanted interactions with other processes. A program may use an inter-process communication facility to send *requests* to other programs; or to receive requests from other programs, process them, and return results in a *response*. A program sending requests is called a *client*; one receiving them is called a *service*; a program may be both a client and a service.

There are many providers of services, not just the system. In particular, many security services are provided by non-system entities, and they may not be fully trusted.

We use an object oriented model: clients use requests to ask services to perform some *operation* on an *object*[2] that the service implements. Services in turn invoke other services to perform the

[2] Another frequently used term for object is resource. In this context, they mean the same thing. A service may implement only a single object, or it may implement many. If many, they may all be of the same kind, or they may be of different kinds.

requested operation. Ultimately, they invoke drivers to write pixels to the screen, bits to the disks, packets to the network, etc.

Basic security model

Computations run on behalf of *principals*; principals may be users or services (and other kinds, to be defined below, but these are the basic ones). A system service exists that can start an initial process and program on behalf of a user after verifying the user's identity and their permission to use the system.

Requests can be in many forms; typical examples are messages sent over a network or inter-process communication mechanism, or APIs that call into the operating system[3].

Services are responsible for securing themselves; i.e., making sure that only authorized principals will have their requests executed. When a service receives a request, it forms the *security context* for that request, uses its *trust policy* to validate all the information in the security context, and then uses it to evaluate its *authorization policy*[4] to decide if the request should be honored. The next few sections expand on this process.

Model components

A *statement* is a collection of data created by a principal; statements can contain other statements. A *claim* is a statement consisting of security relevant information about a principal; a *security token* is a statement containing one or more claims. An important type of claim is the *attribute-value (AV) claim*, stating that a principal has certain attributes; such a claim might be that a user has a certain identity, is a member of a specific group, or has a certain credit limit. A security token might be a list of group memberships for a user.

A *signed statement* is a statement for which an AV claim attesting to the identity of the principal making the statement can be requested from the system; they are particularly interesting when the statement is a security token. The system guarantees that

[3] The request identifies the operation and the object on which it is to be performed (if it's not implicit) and contains any other data needed to perform the operation.

[4] The analogy is to the standard model of interpretation: the policy contains free variables that a bound with reference to the context.

signed statements are tamperproof and the principal's identity is unforgeable[5].

Requests and responses are statements, and they too may be signed[6]. Whenever necessary, the system can guarantee that signed requests and responses are private; i.e., the contents are not accessible to any process except the intended recipient.

A security context is a collection of claims related to a particular request. It can be initialized with the AV claim identifying the sender of a signed request, or by a security token. Security tokens may be received in requests, or returned in responses to requests made to other services; a service whose primary purpose is to do the latter is called a *security token service* (STS). Multiple security contexts may be merged to form a new security context just by taking the union of all their claims.

Trust model

The claims in the security context are validated against the service's *trust policy*. The trust policy for a service defines which of a security token service's claims will be used when evaluating its authorization policy; the service will trust a claim if it deems the service (often an STS) that made the claim authoritative for that claim. Any given STS may (and usually will) be considered by any given service to be authoritative for only a subset of all principals, and, for any principal, only a subset of the possible kinds of AV claims that can apply to that principal; we call this its *authorization scope* with respect to that service. For example, the human resources service for a division of a corporation may be authoritative for AV claims about salaries of division employees, while the division IT department's group membership service is authoritative for AV claims about their group memberships.

There is a kind of claim, which we call a *trust claim*, which defines an authorization scope for a particular STS. The trust policy for a service is a collection of such claims. In addition, authorization scope claims can be in the security context and will be trusted if they were made by an STS that is trusted (i.e., authoritative for

[5] To simplify exposition, we have simply posited that the system can do this, but it should be noted that in Taos both identity and authorization are verified in a uniform way using (its analog to) claims and the trust validation we outline in this paper. I.e., user identity is just an AV claim.

[6] We allow unsigned requests for cases where anonymity is allowed or desired.

them). Note that trust claims are themselves a kind of AV claim: they specify a set of claims for which a service is authoritative and is therefore trusted to make.

Trust policy, in the form of a security token containing trust claims, can be an argument to a request, and also are validated against the service's trust policy. Trust claims that pass validation may be added to the service's trust policy. Trust policies can be combined to create a new trust policy just by taking the union of all their claims.

More complex principals

Principals can be organized into *groups*: a group is a set of users or groups. A group is a kind of principal: a group member is authorized to do anything that the group is authorized to do.

Principals can also be organized into *roles*. A role is a kind of principal: a role member is authorized to anything the role is authorized to do. A role differs from a group in that its membership is tied to an object type and a scope – see the next section.

A principal may be formed from a set of other principals, making an *access token*[7]: a token is authorized to do anything that any principal in the token is authorized to do. Tokens can also be *restricted* by specifying a second set of principals; a restricted token is authorized to do anything that both sets of principals are allowed to do. These constructs allow taking the "or" and "and" of principals (respectively).

Authorization policy

A service may associate with each operation of the service a *permission* that authorizes the operation[8]; the operation is said to *require* the permission.[9] Associated with each object in a service is its authorization policy.[10] An authorization claim for an object specifies a set of principals, and the permission(s) *granted* to that

[7] Often referred to simply as a "token" when the context is clear.
[8] More than one operation may be associated with a given permission.
[9] It is possible, but not encouraged, for an operation to require more than one permission.
[10] More than one object may be associated with a given authorization policy.

set[11]. The set of principals can be specified by a Boolean expression which evaluates to true for all members of the set, where the free variables in the expression are bound to the values of attributes in AV claims in the security context. The authorization policy for an object is a set of such claims.

Objects in a service can be organized into *scopes*: all objects of the same type[12] in the same scope have the same assignment of principals to roles. Assigning scopes simplifies authorization management by removing the need to manage authorization policy for each object individually.

One kind of authorization policy is *role based*: all objects in the service of the same type have the same authorization policy, and the only principals in the authorization policy are roles. With role based authorization, the authorization policy is fixed by the implementation of the service, which "hard codes" the assignment of permissions to roles; authorization is managed by changing the assignment of principals to roles and objects to scopes.

Authorization policy, in the form of a security token containing authorization claims, can be an argument to a request, and also is validated against the service's trust policy. Authorization claims that pass validation are added to the service's authorization policy.

Authorization policies can be combined to create a new authorization policy just by taking the union of all their claims.

Authorization verification

To secure itself, a service utilizes a *reference monitor*: for each request, it asks the reference monitor to decide whether it should grant the request. The reference monitor bases its decision on the security context for the request, the operation requested, the service's trust policy, and the service's authorization policy. (For example, a basic kind of authorization policy could simply specify which principals can perform what operations on its objects; one way to express this is with access control lists on the objects.) Essentially, the trust policy is used to create a *trusted security context* that only has trusted claims, then the authorization policy is treated

[11] Note that the set of principals with a given permission essentially defines a group.

[12] For purposes of this paper, it suffices to define that objects have the same type when they implement the same operations.

like a program to be executed, with the free variables in it assigned values from the trusted security context. If the reference monitor OKs the request, then the service executes the operation, using its own identity to make the requests on any other services or drivers needed to do so.

The model above leads to the following flow for verifying that the authorization policy is satisfied when a service processes a request:

```
Get the operation specified in the request
Combine all the security tokens to create the
  security context
Create the trusted security context by using the
  trust policy to remove untrusted claims
Get authorization policy:
    If only one policy for the service, just return
      it; else:
       Determine the object being referenced by the
         request
       Determine the object's scope
       Determine the object's type
       Get authorization policy for that type in
         that scope
Determine if the requesting principal is given the
  required permissions by the authorization policy:
    If the principal is an access token, take the
      union of the permissions associated with each
      principal in the access token
    If the principal is a restricted token, take the
      intersection of the permissions associated
      with each principal in the restricted token
If the permissions do not include the one required
  for the requested operation return an access
  denied error, else return OK
```

Note that if a service does not have need for flexible configuration of authorization policy and wants the ultimate in efficiency, then it can associate a role with each operation, and have the implementation of each operation simply check whether the requesting principal is that role (or an access token that contains that role).

Conclusions

We have briefly described a conceptual model for authorization for web services. If one contrasts it with "more traditional" models, the more interesting differences include:

- authorization based not just on user identity and group memberships but on attributes of users

- support for partial trust on attributes as well as user identity and group memberships
- trust and authorization policy can be arguments to requests from untrusted clients, as long as they originate with parties trusted to set such policy

Finally, this model isn't really tied to web services – it could be used in other distributed systems contexts where the features that differentiate it from the more traditional model are needed, just as web services need them.

Acknowledgements

We would like to acknowledge the support given to the work that led to this paper by Dave Aucsmith, Doug Bayer, Peter Biddle, Blair Dillaway, Mike Nash, David Treadwell, and Robert Wahbe.

References

[Lamp92]
LAMPSON, B., ABADI, M., BURROWS, M. AND WOBBER E., 'Authentication in distributed systems: Theory and practice,' *ACM Trans. Computer Systems* vol. 10, no. 4, Nov. 1992, pp. 265-310. A preliminary version is in the *Proc. 13th ACM Symposium on Operating Systems Principles*.

[LAMP74]
LAMPSON, B., 'Protection,' *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971. Reprinted in *ACM Operating Systems Review*, vol. 8, no. 1, Jan. 1974, pp. 18-24.

[NEED78]
NEEDHAM R.M. AND SCHROEDER, M.D., 'Using encryption for authentication in large networks of computers,' *Comm. ACM*, vol. 21, no. 12, Dec. 1978.

[SDSI]
LAMPSON, B. AND RIVEST, R., SDSI – A Simple Distributed Security Infrastructure, http://theory.lcs.mit.edu/~cis/sdsi.html, 1996.

[WS02]
IBM, MICROSOFT. Security in a Web Services World: A Proposed Architecture and Roadmap, 2002. http://msdn.microsoft.com/library/en-us/dnwssecur/html/securitywhitepaper.asp