Trust Management for Web Services

Scott D. Stoller
Computer Science Department
Stony Brook University
stoller@cs.stonybrook.edu

***Abstract*—Service-Oriented Architecture (SOA) is increasingly used in enterprise information systems, particularly in the form of Web Services. This paper describes a practical trust management system for Web Services that allows information in databases to be used seamlessly and efficiently in trust management policies.**

I. Introduction

Service-Oriented Architecture (SOA) is increasingly used in enterprise information systems, particularly in the form of Web Services. However, the access control mechanisms built into current Web Services frameworks cannot express the complex security policies of large organizations. As a result, much of the access control for enterprise applications is implemented in application-specific code. Security policies implemented this way are more difficult and therefore more expensive to write, read, verify, and maintain, compared to security policies in simpler, more declarative policy languages. Therefore, we advocate extending the Web Services framework with an application-independent access control mechanism that is capable of expressing enterprise security policies.

A crucial requirement for the design of such a system is support for the decentralized nature of enterprise systems. The security policy and the information it uses come from multiple sources, and the policy must specify which sources are trusted for which decisions and information. This observation motivated work on trust management [1]. Another crucial requirement for the design is convenient and efficient support for policies that use information in relational databases.

No existing trust management framework satisfies these two requirements. Most trust management frameworks are not designed to integrate with Web Services or databases. PeerTrust [2] is a trust management framework for the Semantic Web, but it is not well suited for applications that heavily use databases. iAccess [3] is a semantics-based trust management framework with sophisticated trust negotiation, but use of information in databases is not considered in its design.

This paper describes a trust management and trust negotiation framework that satisfies these two requirements. The design builds on the trust management framework for relational databases described in [4], which extends the original trust management framework for relational databases in [5]. A key feature of the approach is that the policy language is a relatively small extension of SQL. This has multiple benefits: data stored in the database can conveniently and efficiently be used in the policy; the implementation gets the benefits of decades of work on efficient evaluation of SQL queries; and the learning curve is small for people who already know SQL. In contrast, the trust management frameworks mentioned in the previous paragraph, including PeerTrust and iAccess, have policy languages based on Datalog, which is elegant but much less widely used.

This work was supported in part by ONR under Grants N00014-07-1-0928 and N00014-09-1-0651, NSF under Grants CCF-0613913, CNS-0627447, and CNS-0831298, and AFOSR under Grant FA0550-09-1-0481.

II. System Architecture

A common architecture for an enterprise Web application includes a Web services engine (server), on which application Web services are deployed. Each Web service offers methods that are invoked by clients using a remote invocation protocol, such as SOAP. Application data is stored in a DBMS. We assume the system has a public-key infrastructure (PKI). We add two pieces to this typical architecture.

A *trust management service* deployed on the server provides methods for using and managing (administering) the trust management system. Informally, we classify the methods in the trust management service API into two groups. The first group contains methods useful for clients that want to access application Web services; these clients use the trust management service to obtain the necessary permissions. The second group contains methods useful for clients that want to manage (administer) the trust management policy. (Methods in the first group are also useful for these clients). These methods are used to manage the trust management policies for all the Web Services, including the trust management service itself. In this paper, the API is described in an informal notation; in our prototype, the API is defined in WSDL.

An *interceptor* (handler) in the Web services engine enforces the trust management policy. The engine invokes the interceptor on all incoming requests to invoke methods of the Web service (e.g., SOAP request messages). The interceptor queries the trust management policy and tells the engine to permit or deny the request as appropriate.

Our prototype implementation uses Apache Axis2/Java on Apache Tomcat, and MySQL (thanks to Raveesh Ahuja for implementing the prototype). As a case study, we translated the trust management policy for electronic health records in [6] into our policy language. It is the largest and most realistic formal trust management policy that we have seen. The translation demonstrates that realistic trust management policies can be expressed in our framework.

III. Policy Representation

The trust management policy for a Web service is defined by the contents of SQL views called *permission views*. Permission views may refer to regular tables and special tables called *certificate tables*, or "certtables" for short. Permission views and certtables are described in detail below.

A. Permission View

A permission view is, conceptually, a function from the request details to the access control decision (permit or deny). Let $S.m$ denote a method m provided by a Web service S. The "arguments" to the permission view for method $S.m$ are stored in a table named `request_`S`_`m, which contains columns `invoker` (invoker's public key) and `invokerDN` (invoker's distinguished name), plus columns corresponding to the arguments of method $S.m$. The interceptor stores information about the current request in this table before evaluating the permission view, converting the XML representation of the arguments to an appropriate SQL representation. If evaluation of the permission view returns a non-empty result set, the request is permitted; otherwise, it is denied. Views are defined using the trust management service's `createView` method. Its prototype is `createView(name, viewDef)`, where `name` is the view name, and `viewDef` is the body of an SQL view definition.

For example, consider a Web service named `HRsvc` that provides access to electronic health records. The service defines a method `agentViewItem(byte[] patient, int itemID)` that allows a patient's agent (e.g., spouse) to view an item in the patient's health records. The `agent` table contains a record with s in the `subject` column and p in the `patient` column if s is an agent for p. A sample permission view for this method is as follows, where `requestAVI` abbreviates `request_HRsvc_agentViewItem`.

```
createView("permView_HRsvc_agentViewItem",
"SELECT * FROM agent, requestAVI
 WHERE agent.subject = requestAVI.invoker
       &&
       agent.patient = requestAVI.patient
 LIMIT 1")
```

This says that an invocation of `agentViewItem` is permitted if the invoker is in the set of agents of the patient identified by the `patient` argument of `agentViewItem`.

B. Certificate Table (Certtable)

A *certificate table*, abbreviated as *certtable*, is a special kind of table that stores information from specified trusted sources (issuers) [4]. Only information obtained in signed X.509 attribute certificates can be inserted in certtables. An *attribute certificate* contains a list of attribute-value pairs, the issuer's public key, and a digital signature. One of the attributes must be `subject`; the other attributes provide information about the subject.

A certtable is defined by invoking the trust management service's method `createCerttable(name, colDefs, constraint, issuers, fetchFrom, releaseTo)`, where `name` is the name of the certtable, `colDefs` is a comma-separated list of column definitions of the form *name type* (as in SQL), `constraint` is a Boolean expression of the form allowed in the `check` clause in an SQL `create table` statement, and `issuers` specifies the trusted sources for information stored in this table (in other words, only information from those sources may be inserted in this table). `issuers` may be a public key, identifying a specific issuer, or a query of the form `select` *column* `from` *ctv*, where *ctv* is the name of a certtable, table, or view. The second form means that the users whose public keys are returned by the query are trusted issuers for this certtable. The `fetchFrom` and `releaseTo` arguments are described in Section VI. When the allowed issuers are specified using a certtable, table, or view *ctv*, if records for an issuer are removed from *ctv*, certificates issued by that issuer are automatically removed from the certtable.

Every certtable contains the following columns, even if they are not explicitly declared: `subject`, `subjectDN`, `issuer`, `expiration`, and `certificate`. The `certificate` column contains the X.509 certificate from which the information in the record was obtained.

For example, if doctors are trusted issuers for information about agents, the `agent` certtable could be defined by (the `certType` attribute makes the certificate's purpose explicit)

```
createCerttable(
  name="agent",
  colDefs="certType varchar(30),
           patient varchar(1000)",
  constraint="certType='agent'",
  issuers="select subject from doctor",
  fetchFrom="issuer",
  releaseTo="GP for same patient" )
```

IV. Trust Management Service API: Methods for Policy Administrators

Figure 1 summarizes methods in the trust management service API that are useful (in addition to the methods in Section V) for managing the trust management policy.

The `grant` method in Figure 1 is used to grant permissions for invoking methods of the trust management service. Note that the `grant` method is not used to grant permissions for invoking methods of application Web services; that is accomplished by defining appropriate permission views and

```
createCerttable(name, colDefs,
   constraint, issuers, fetchFrom,
   releaseTo)
createView(name, viewDef)
grant(operation, resource, grantees,
   grantName)
revoke(grantName)
setPermView(service, method, view)
```

Fig. 1. Trust Management Service API: Methods for Policy Administrators

Operation	Resource
`create`	`certtable` or `view`
`delete`	name of certtable
`insert`	name of certtable
`grant`	pair: operation and resource
`requestPerm`	none
`revoke`	name of grant
`setPermView`	pair: names of service and method

Fig. 2. Operations and resources used in `grant` table

updating contents of certtables or tables used in permission views. For methods of the trust management service, the permission views are fixed by the design of the framework; granting permissions for those methods is accomplished by updating the contents of a table, called the `grant` table, used in those views. Administrators update the contents of the `grant` table using the `grant` and `revoke` methods.

The `grant` table has the following columns: `operation` (operation for which permission is granted), `resource` (resource on which the operation is permitted), `grantees` (users to whom permission is granted), and `grantName` (unique identifier for this record). We refer to an operation and resource together as a "permission" or "privilege". The use of operation names, instead of method names, in permissions, is a convenient abstraction. `grantees` is the name of a certtable, table, or view whose `subject` column contains the identities (public keys) of the grantees. Figure 2 shows the allowed values of `operation` and the corresponding allowed values of `resource`. In resources, the special value "*", called *wildcard*, can be used to represent all (other) allowed values. The resource for the `grant` operation is a pair containing an operation and a resource; this is a recursive definition.

`createCerttable` is described in Section III-B. The permission view for `createCerttable` checks that (*i.e.*, returns a non-empty result set if) the invoker has permission for `create` on `certtable`. `createCerttable` updates the `grant` table to give the invoker appropriate permissions on the new certtable. All certtables are owned by a database account associated with the trust management service, not by the invoker of `createCerttable` (who might not even have a database account); this is also true for views created using `createView`.

`createView` is described in Section III-A. The permission view for `createView` checks that the invoker has permission for `create` on `view`. `createView` updates the `grant` table to give the invoker `select` and `grant` permissions on the new view.

`grant` inserts a record in the `grant` table. The arguments of this method correspond directly to the columns of the `grant` table. The permission view for `grant` checks that the invoker has permission for `grant` on the pair containing the specified operation and resource. `grant` updates the `grant` table to give the invoker `revoke` permission on the new grant.

`revoke` removes the record with the specified name in the `grantName` column from the `grant` table. The permission view for `revoke` checks that the invoker has permission for `revoke` on the named grant.

`setPermView` sets the named view to be the permission view for the specified method of the specified application Web service, by updating a table maintained by the trust management service. The permission view for `setPermView` checks that the invoker has permission for `setPermView` on the pair containing the specified service and method.

```
insertAttribCert(cert, certtable)
insertPKcert(cert, certtable)
deleteCert(certtable, constraint)
getCert(col, val, colDefs, constraint)
requestPerm(service, method)
```

Fig. 3. Trust Management Service API: Methods for Application Clients (and Policy Administrators)

V. Trust Management Service API: Methods for Application Clients

Figure 3 lists methods in the trust management API that are useful for application clients (and policy administrators).

`insertAttribCert` inserts a record in the specified certtable. The new record contains the attribute values from the specified X.509 attribute certificate *cert*; the certificate itself is stored in the `certificate` column. The permission view for `insertAttribCert` checks that the invoker has `insert` permission on the specified certtable. The insertion succeeds if (1) *cert* has a valid signature, (2) *cert* contains attributes corresponding to all of the columns in the certtable (the certificate may contain other attributes as well), (3) *cert*'s issuer is allowed by the certtable's issuer specification, and (4) the attribute values in *cert* satisfy the certtable's `constraint`. If the `certtable` argument is the empty string, the certificate is inserted in all certtables on which the invoker has `insert` permission and for which the above conditions are satisfied.

`insertPKcert` is similar to `insertAttribCert`, except that the argument is an X.509 public-key certificate, which is treated as an attribute certificate containing only `subject` (subject's public key), `subjectDN` (subject's distinguished name), `issuer`, and `expiration` attributes.

`deleteCert` deletes records satisfying the specified constraint from the specified certtable. The constraint is a Boolean expression of the form allowed in SQL `delete` statements. The permission view checks that the invoker has permission for `delete` on the specified certtable.

`getCert(`*col*`,` *val*`,` *colDefs*`,` *constraint*`)` returns all certificates, stored in any certtable, that satisfy the criteria indicated by the method arguments. Specifically, it returns certificates that (1) have an attribute named *col* with value *val*, (2) contain attributes corresponding to all of the columns in *colDefs*, (3) satisfy the *constraint* on the attribute values, and (4) are releasable to the invoker, as defined in Section VI.

`requestPerm` attempts to provide the invoker with permission to invoke the specified method of the specified service, by fetching relevant attribute certificates, using the algorithm in Section VI. The permission view checks that the invoker has `requestPerm`.

VI. Certificate Fetching and Trust Negotiation

Users can obtain desired permissions by explicitly inserting certificates in certtables. However, relying on users to do this is usually impractical. Our trust management service supports automated fetching of certificates from servers indicated in the policy. This is sometimes called *credential discovery* [7].

The `fetchFrom` argument of `createCerttable` (see Figure 1) specifies a user from whom the trust manager should request certificates when a desired certificate is not already present in the certtable. The legal values are: `""` (do not fetch certificates for this certtable); `"issuer"` (request certificates from the users specified by the `issuers` argument for this certtable); and `subject` (request certificates from the subject of the desired certificate). The trust management service fetches certificates from a user by invoking the `getCert` method on the specified user's *home server*. In our prototype, a user's home server is named in the user's public-key certificate; alternatively, a directory service could be used.

The `releaseTo` argument of `createCerttable` defines the *release policy* for the certtable. It specifies the set of users to whom certificates in the certtable may be released via (*i.e.*, returned from) `getCert`. The legal values of `releaseTo`, and the sets of users they denote, are: `""` (no one), `public` (everyone), a public key (that user), *ctv* (the name of a certtable or view, denoting the subjects of records therein), and *ctv* `for same` *col* (the subjects of records r in *ctv* such that column *col* of r has the same value as attribute r in the certificate being considered for release).

Examples of `fetchFrom` and `releaseto` appear in the definition of the `agent` certtable in Section III-B. `releaseTo="GP for same patient"` means that agent certificates for a patient may be released to the patient's general practitioner (GP). This assumes the certtable `GP` contains a record with `subject` = c and `patient` = p if clinician c is patient p's general practitioner.

The trust management service uses certificate fetching—in the form of calls to `getCert`—in multiple places. (1) In the algorithms for `insertAttribCert` and `insertPKcert`, if the issuers of the target certtable t are specified by a certtable or view *ctv*, and there is no record in *ctv* for the issuer i of the certificate being inserted, certificate fetching is used to try to obtain certificates about i that establish that i is a trusted issuer for t. This is done recursively: if *ctv* is a certtable, certificates are fetched for its `issuers` certtable (if necessary), and so on. (2) In the algorithm for `getCert` itself, if a candidate certificate cannot be released because there is no suitable record for the invoker in a certtable t used in the relevant `releaseTo` policy, the algorithm calls `getCert` to try to obtain such a certificate. This can lead to a distributed "chain reaction" of calls to `getCert`. Such interactions are usually called *trust negotiation*. (3) Certificate fetching is used in the algorithm for `requestPerm`, as described next.

`requestPerm(service, method)` works as follows. If the specified service is the trust management service `TMsvc`, let T be the set of certtables that appear in the `grantees` column of any record in the `grant` table with the specified method in the `operation` column; otherwise, let T be the set of certtables t such that t or a view that depends on t appears in the permission view for the specified method of the specified service.

For each certtable t in T, the trust management service tries to obtain certificates whose `subject` is the invoker and that can be inserted in certtable t. It does this by invoking `getCert(subject,` u`, colDefs(`t`), constraint(`t`))` on the home servers of users indicated by `fetchFrom(`t`)`, where `colDefs(`t`)`, `constraint(`t`)`, and `fetchFrom(`t`)` return the `colDefs`, `constraint`, and `fetchFrom` arguments, respectively, from t's definition. Furthermore, if it receives a certificate c that cannot be inserted in the target certtable t because there is no record for c's issuer i in the certtable or view (if any) in `issuers(`t`)`, it uses certificate fetching as described above for `insertAttribCert` to try to establish that i is an issuer for t.

VII. Nested Service Calls

SOAs often lead to *nested service calls*: a server may invoke other services while processing a request, those other services may utilize yet other services, *etc*. To support secure processing of nested service calls, our system keeps track of the *calling context* of each request—a sequence of public keys identifying the services in the chain of nested calls that led to this request. We extend `request` tables with a `context` column, and we extend the interceptor for incoming calls to extract the calling context from the SOAP header and store it in the `context` column of the `request` table. The calling context can be used freely in permission views.

References

[1] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, Nov. 1992.

[2] W. Nejdl, D. Olmedilla, and M. Winslett, "Peertrust: Automated trust negotiation for peers on the semantic Web," in *Proc. 2004 Workshop on Secure Data Management (SDM)*, ser. Lecture Notes in Computer Science, vol. 3178. Springer-Verlag, 2004, pp. 118–132.

[3] H. Koshutanski and F. Massacci, "Interactive access control for autonomic systems: from theory to implementation," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 3, Aug. 2008.

[4] S. D. Stoller, "Trust management and trust negotiation in an extension of SQL," in *Proc. 4th International Symposium on Trustworthy Global Computing (TGC 2008)*, ser. Lecture Notes in Computer Science, vol. 5474. Springer-Verlag, Nov. 2009, pp. 186–200.

[5] S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Trust management services in relational databases," in *Proc. 2nd ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS '07)*. ACM, 2007, pp. 149–160.

[6] M. Y. Becker, "Cassandra: Flexible trust management and its application to electronic health records," Ph.D. dissertation, University of Cambridge, Oct. 2005.

[7] N. Li, W. H. Winsborough, and J. C. Mitchell, "Distributed credential chain discovery in trust management," *Journal of Computer Security*, vol. 11, no. 1, pp. 35–86, 2003.