Trust Negotiation as an Authorization Service for Web Services

Lars Olson, Marianne Winslett
University of Illinois at Urbana-Champaign
{leolson1,winslett}@cs.uiuc.edu

Gianluca Tonti
Università di Bologna, Italy

Nathan Seeley
Brigham Young University

Andrzej Uszok, Jeffrey Bradshaw
Institute for Human and Machine Cognition, Pensacola, Florida

Abstract

Like other open computing environments, web services need a scalable method of determining authorized users. We present desiderata for authorization facilities for web services, and analyze potential ways of satisfying them. We propose a third-party authorization system for web services based on trust negotiation, discuss its implementation using the TrustBuilder runtime system for trust negotiation, and present performance results from a stock trading application.

1 Introduction

As web services become more widely used in open distributed systems, the need for security enforcement is apparent. The security needs can potentially include guarantees of confidentiality, integrity, authentication, authorization, availability, auditing and accountability. In this paper, we will focus on the authorization-related needs of applications relying on web services. The desiderata for such an authorization facility will depend on the application, and may include:

Openness: The authorization facility should allow qualified strangers to obtain authorization to access resources. An open system cannot rely on traditional distributed systems authorization facilities based on identity. For example, authorization facilities based on logins and passwords will not be suitable, because strangers will not have a login at the service they wish to access.

Corporate- and Shibboleth-style authorization services for an organization rely on X.509 identity certificates (or a similar authentication substrate) backed up by the organization's LDAP servers, which know about each identity's attributes (e.g., employee names and positions). This architecture allows the LDAP servers to store an arbitrary set of properties about the identities within the organization, allowing support for attribute-based authorization. However, this architecture is still not sufficiently open to meet our requirements. First, the set of attributes that a client may possess is predetermined and static. In an open system, one cannot predict all the resources a client might wish to access, nor the associated authorization requirements. Second, the client must have a previous relationship with an organization that records the client's attributes. Further, that organization must have a previously established trust relationship with the owner of the resource, specifying that the resource owner is willing to accept attribute attestations from that organization. In an open system, those relationships may not exist.

The authorization service should support the establishment of bilateral trust. Traditional distributed system security assumes that the client trusts the server *a priori*, but the server does not trust the client. For a truly open system, the client also needs to have a way to ask the server to prove that it can be trusted.

Ease of management: The authorization service must be easy to deploy, both for initial setup and subsequent maintenance. Legacy applications must be able to use the new authorization system. Application owners need support tools to help them understand their own authorization policies and understand the effect of proposed updates to their policies.

Confidentiality: First, the authorization service should allow users to retain control over the disclosure of their attributes to other parties. Similarly, resource owners should retain control over the disclosure of the authorization policies for their resources, as those policies may be sensitive. Second, other parties should not be able to eavesdrop and learn about disclosed attributes or sensitive service requests. Third, portions of the run-time authorization conversation must be kept private to prevent security breaches (e.g., passwords must not be transmitted in the clear).

Universality: A general-purpose authorization system should not impose excessive restrictions on the layers be-

neath it. It should be capable of supporting a variety of authorization policy languages, since no single language can meet the needs of all resource owners equally well. It should also work with many different kinds of credential and identity systems. For example, it should not require all attributes to be certified via X.509 certificates (though a resource owner can impose such a condition). Ideally, if a resource owner's policy deems an information source to be trustworthy with respect to a particular attribute, then the system should be able to accept any form of verifiable, unforgeable, nonrepudiable attribute statements from that source.

Integrity: It must not be possible to corrupt a message undetectably during the authorization conversation.

Availability: The authorization service should be highly available, as the applications that rely on it will become unavailable if the authorization service is unavailable. First, the authorization service must scale well as the number of resource owners and potential and actual users increases. This implies that the authorization service should not implement identity-based solutions (e.g., local accounts for all potential users), as they impose large management overheads as the number of potential users increases. It is impractical to expect one organization to closely track membership changes in another organization. Second, the authorization service should react gracefully to heavy loads and be resilient against denial of service attacks.

Auditing and accountability: Identity-based authorization systems can use tamper-resistant logs to provide strong guarantees of accountability and support after-the-fact audits. This is more of a challenge in attribute-based trust management systems. If a client presents a certificate, then in practice that certificate will contain a unique identity for that client *within the issuing domain.* If the client subsequently misbehaves, it may be hard to obtain cooperation from the issuing domain to track down the client and hold her accountable for her behavior. Most proposals for anonymous credentials do not include a means of subsequently identifying the client if necessary.

Autonomy: First, each resource owner should have the right to declare its own authorization policy for access to that resource. For example, clients should not be expected to divulge their attributes without possibly requesting proofs of trustworthiness from the recipient of that information. Second, negotiating parties may have a variety of strategic goals. For example, a client might want to get access to the resource as quickly as possible, while the resource owner might want to collect as much information about the client's attributes as possible before granting access, for marketing purposes. The authorization service should allow its clients and resource owners reasonable leeway in pursuing their own strategic agendas. Third, trust establishment approaches that require participants to adhere to a rigid algorithm are relatively easy to attack. The attacker can extract irrelevant information from the other party, crash the other party, or launch a denial of service attack by not following the algorithm. Increased autonomy in the runtime procedures appears to lessen vulnerability to attack.

The web services community has generated dozens of draft standards related to authorization and other aspects of security, such as WS-Policy, SAML, and XMLDSIG. From the proliferation of standards, one could be forgiven for thinking that web services security is a solved problem. In reality, however, each standard specifies a small building block for the overall security edifice. An authorization service designer needs to choose which standards to use internally and for its input and output.

While the desiderata are surely incomplete (we cannot envisage the authorization needs of all future applications), they still help us to evaluate the suitability of authorization approaches for web services. We immediately see that traditional authorization solutions—such as RADIUS (www.gnu.org/software/radius/radius.html), Kerberos (web.mit.edu/kerberos/www), and identity- or organization-based public key infrastructures (PKIs)—do not meet the scalability and openness desiderata, as they require authorized users (or their organizations) to be known *a priori*. Instead, we advocate the development of a security middleware service that employs *trust negotiation*, an attribute-based authorization approach designed for open systems. In trust negotiation, access policies for resources in the system are written as declarative specifications of the attributes that authorized users must possess. Entities in the system possess digital credentials issued by third parties that attest to their attributes, along with policies that limit access to their sensitive resources and credentials. The trust negotiation process allows credentials and policies to be disclosed between parties in a bilateral and iterative manner that incrementally establishes trust. As explained in the next section, authorization services based on trust negotiation have the potential to satisfy all of the desiderata listed above.

Our proposed authorization approach uses trust negotiation to broker access tokens for web services deployed in the system. In our approach, web services handlers [14] are used to detect service requests for which these types of access tokens are needed. After detection, the handler invokes a trust negotiation facility to negotiate for access to the specified service. Because these handlers are interposed between the human user and the deployed services, negotiations take place automatically, without requiring user intervention. In our implementation of this architecture, we leverage the TrustBuilder framework for automated trust negotiation and use existing web services standards such as XMLDSIG to encode and process the access tokens gener-

ated after successful negotiations take place.

In the remainder of the paper, Section 2 explains how trust negotiation has the potential to satisfy the desiderata. Section 3 describes a web service for stock purchases and its authorization needs. Section 4 presents performance measurements of our third-party authorization service based on TrustBuilder, and Section 5 describes related efforts.

2 Trust Negotiation

For a verifiable, unforgeable, nonrepudiable proof that an entity possesses a certain property, trust negotiation relies on digital credentials issued by third parties. For example, a company could use X.509 attribute certificates for digital employee IDs that list employee numbers, names, and departments. At run time, an entity mentioned in a credential can prove that it is the mentioned entity, typically by demonstrating knowledge of a private key corresponding to a public key that appears in the credential. Trust negotiation can use any kinds of digital credentials, including anonymous credentials.

When an organization wishes to share a resource with outsiders, it can create an authorization policy that describes the attributes that authorized users must possess. For example, a policy might specify that access is limited to IBM employees, companies that supply WalMart with merchandise, parents of kindergarteners in Texas, or nurses working for doctors who are treating a particular patient. Often the credentials themselves will contain sensitive information (e.g., a social security number in a driver's license), in which case they should also be protected by authorization policies. When credentials are sensitive, their disclosure may require the recipient to produce certain credentials first. During a trust negotiation, the negotiating parties iteratively disclose increasingly more sensitive credentials, until either the authorization policy is satisfied or one party gives up.

Figure 1 shows an example trust negotiation. Alice has a broker ID credential, protected by a policy that requires a certificate from the SEC showing that the holder is authorized as a stock exchange. Bob is a stock exchange that offers an internet-based exchange service to stock brokers. Bob's authorization policy for the service requires that the customer present a current broker ID from a recognized brokerage firm. A negotiation is triggered when Alice tries to access the service. Bob responds by sending Alice the authorization policy that protects this service. Alice can satisfy this policy with her broker ID, but she is not willing to disclose that credential to Bob, because its authorization policy is not yet satisfied. Bob has no way to know her policy *a priori*, so Alice sends her broker ID policy to him. Bob can satisfy this policy; he sends Alice his certification issued by the SEC, which he does not consider sensitive. Along with that credential, he sends a proof that he owns the certificate. The policy that protects Alice's broker ID is now satisfied, so she sends it to Bob, along with a proof that she owns it. The authorization policy for the exchange service is now satisfied, so Bob grants Alice access.

Authorization services based on trust negotiation have the potential to satisfy all of the desiderata listed in the previous section:

Openness: Trust negotiation easily satisfies the openness requirements. Attribute-based authorization policies only need to be modified when the *authorization criteria* for a resource itself changes, and these criteria need not involve frequently-changing lists of identities of individuals or organizations. The bilateral, iterative nature of trust negotiation allows both parties to establish trust in the same way.

Ease of management: Current policy engines do not satisfy this desideratum, though future trust negotiation systems can do so. Additionally, the authorization service described in this paper can be adapted to legacy trust-unaware applications, thus lowering the barrier for adoption.

Confidentiality of attributes and policies: Because credentials can themselves be considered sensitive resources and be protected by policies, trust negotiation supports confidentiality for sensitive attributes. The policies themselves may be sensitive, so that directly exchanging policies is not desirable. Trust negotiation supports protecting sensitive policies in the same manner as sensitive attributes.

Universality: Trust negotiation does not require the use of one particular kind of digital credentials, and can be used with anonymous credentials and zero-knowledge proofs of properties [15]. If the need arises for an unrecognized credential type, either party should be able to (for instance) download the proper credential parsing and verification package.

Integrity and confidentiality: Trust negotiations can be conducted over a secure channel, such as an SSL/TLS connection, to provide integrity and confidentiality for the conversation. (This connection is not based on any trust attributes; it only provides confidentiality against eavesdroppers, and can be set up using one-time-use self-signed certificates.)

Availability: Researchers have proposed ways to make implementations of trust negotiation resilient against attacks, especially denial of service (DoS) attacks [17]. Current trust negotiation implementations have not made scalability a major goal; future implementations should do so.

Auditing and accountability: Trust negotiation activity can be logged in a tamper-evident logging facility. However, in the case of zero-knowledge proofs of properties and anonymous credentials, the log may not be very helpful in tracking down individuals who behave inappropriately.

Autonomy: It is possible to implement trust negotiation in a manner that preserves the autonomy of parties in making strategic run-time decisions [22]. Again, current imple-

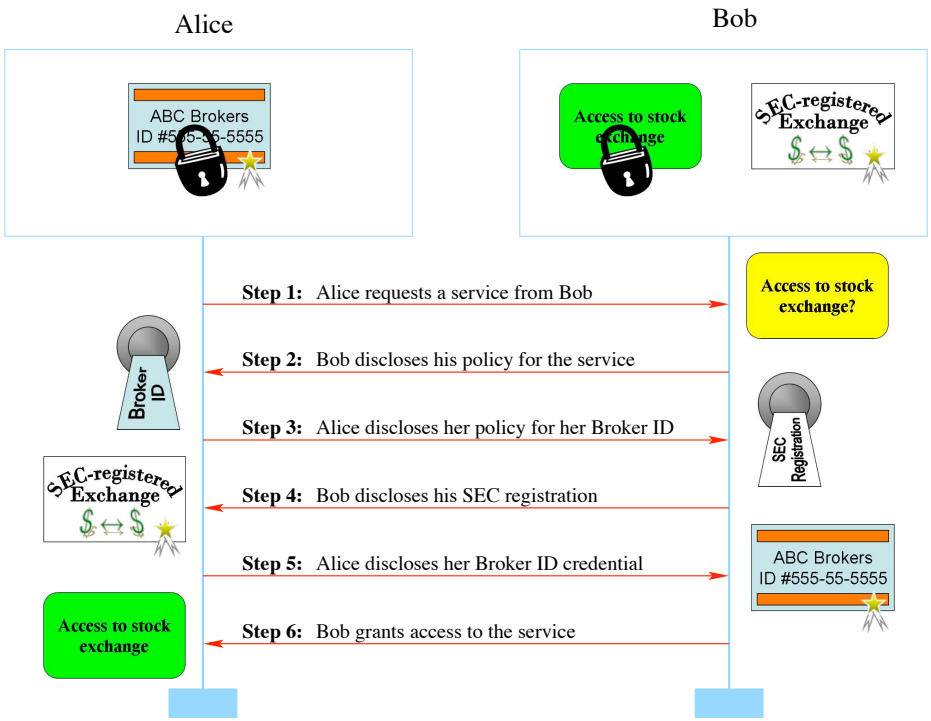


Figure 1. Example of trust negotiation

mentations have not fully addressed this.

Several trust negotiation prototypes have been built to date, including Cassandra [2], idemix [9], QCM [7], SD3 [10], PeerTrust [16], SPKI/SDSI [6], TrustBuilder [20], and Trust-X [3], with others proposed or under development. Since trust negotiation is a new idea, these are all early prototypes that satisfy only a few of the desiderata. Their current implementations either have relatively limited functionality or rely on policy languages or credential types that are not appropriate for our purposes, with two exceptions: TrustBuilder and Trust-X.

TrustBuilder [20] supports the iterative establishment of bilateral trust and provides confidentiality for sensitive attributes. TrustBuilder does not itself provide confidentiality of conversations, but can be run over SSL/TLS for that purpose. TrustBuilder has been tested with two different policy engines, one based on the RT policy language and the other on IBM's Trust Engine (TE), which uses IBM's Trust Policy Language (TPL). The publicly available version of TrustBuilder supports IBM's TE, which imposes its own limitations on the system, including a reliance on X.509 credentials. In practice, this and the other restrictions in the currently available version mean that TrustBuilder has limited autonomy for users (only an eager strategy for disclosures is currently supported), though planned support for additional policy engines will relieve this problem. An unreleased version of TrustBuilder sports increased resilience against attack [17], while an earlier version is available for download over the internet at http://isrl.cs.byu.edu. TrustBuilder has been applied to distributed authorization scenarios using Java-RMI, TLS, and HTTP proxies (among others).

Trust-X [3] offers an XML-based framework for trust negotiation that supports the establishment of bilateral trust, provides confidentiality for sensitive attributes and policies, and provides tokens to represent the successful outcome of negotiations. Trust-X supports additional useful kinds of policies, such as P3P privacy policies to limit redissemination of information disclosed during negotiation. Trust-X supports the autonomy of negotiating parties in making strategic decisions, by providing five different kinds of negotiating strategies that differ in how suspicious the party is of its partner.

Our implementation of an authorization service for web services uses the TrustBuilder prototype. We expect that similar design issues and considerations would hold for an implementation based on Trust-X.

3 Stock Trading Scenario and Authorization Architecture

Suppose that we are operating a stock exchange over a web services platform. Careful auditing, timely service, and message integrity are necessary for such a system. Messages containing stock quotations and other information re-

lated to stock purchases and sales should be readable by independent auditors who track the operations, discouraging unethical or illegal transactions like insider trading. Figure 2(a) shows an example interaction where the client wishes to purchase 500 shares of stock from company ABC. The server makes the purchase at the current price, makes the money transfer, and reports back to the client that the transaction was successful. An open authorization system is appropriate for such a scenario, so that broker companies retain control over the certification of their own employees without requiring the exchange server itself to register and keep track of individual employees. Each broker company can decide who its employees are, issue them appropriate credentials, and define appropriate policies to protect its own credentials from inappropriate disclosure.

One potential architecture for web services authorization is to embed authorization facilities directly in web services clients and servers. However, this would involve duplicating the authorization-related code and trust negotiation functionality in every web service and client that needs it. This would require future upgrades to the authorization package to be installed in all clients and servers, and would clutter the main code of the web service with authorization-related considerations.

To avoid these problems, we chose to build a stand-alone third party authorization service that can be shared by multiple web services. To provide these properties, and to allow legacy applications to be adapted to the authorization service, we defined handlers [14] to implement the trust negotiation and authorization process. A handler is a stand-alone piece of code that does text processing on a SOAP message that has been created by a web service client or server. The possible types of text processing range from simply logging messages that pass through, to adding elements to the SOAP message, to interrupting delivery of messages. Several handlers can be combined in a manner reminiscent of a network stack, where each layer can be defined independently. This way, the main client and server code can focus on the core functionality of the web service, leaving such issues as authorization and logging as abstractions that can be managed separately. Multiple clients or servers can include their respective authorization handlers in the handler chains, reusing the same authorization code rather than duplicating effort. This approach also gives us the added benefit of adding trust-negotiation capability to legacy applications, with very little modification of the applications. We define one handler to run on the client side (on the same machine as the client code), and another for the server side (with the server code).

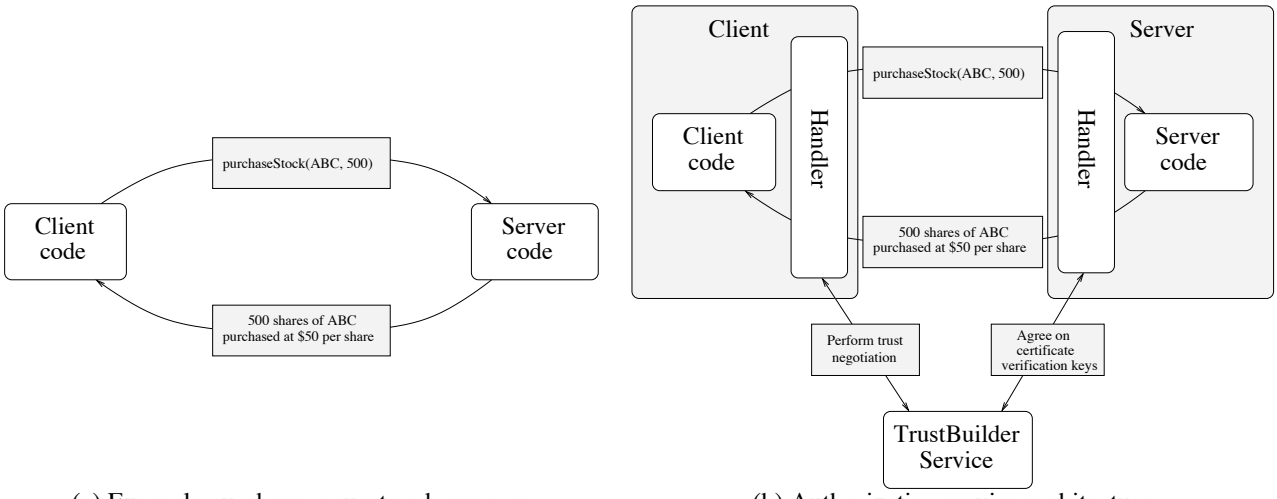
One design option is to embed the authorization-related software in the web service's handler, so that the stock trading client carries out a trust negotiation directly with the handler. We envision, however, that the stock trading service could be part of a larger suite of services, all of which could require a trust negotiation to determine authorization. Thus there are advantages to separating out the trust negotiation software from the server-side handler itself. If the stock exchange expects to have a high volume of authorization requests, then the need for timely response to quotation requests suggests that it might be desirable to run the authorization software as its own dedicated server, possibly on its own dedicated hardware, with the stock exchange's certificates and policies cached locally. If needed, the authorization service can be replicated for availability and to increase scalability. We chose to embed the client-related authorization code, which primarily consists of the trust negotiation module, in the client's handler.

Our final design is shown in Figure 2(b). The stock market server agrees with the authorization server on a long-term cryptographic key to verify the validity of the messages it receives. The client makes a request to purchase stock, and the client's handler detects that this request requires authorization. The client's handler contacts the stock exchange's authorization server and carries out a trust negotiation with the authorization server. The client also generates a public/private key pair for the stock server to verify the client's messages. If the negotiation succeeds in establishing trust, the client's handler is given a signed trust token that contains the client's public key and records the authorization that was granted, namely, that the client is allowed to purchase stocks on the exchange. The client's handler then transmits the client's purchase request to the server, with the token attached to the message. The server's handler verifies the token with the authorization server's key, verifies the rest of the message with the client's key (embedded in the token), and executes the stock purchase operation if the token is valid for purchase operations performed by that client.

The application server must trust the authorization server on the following points: (1) the authorization server enforces the correct policy protecting the application server, which it certifies by including the client's authenticated role in the trust token; (2) it verifies the client's credentials correctly (including revocation checks); (3) it protects the cryptographic key for signing the trust tokens; (4) it does not give the trust token a lifetime longer than any of the client's credentials used to gain authorization; and (5) it revokes the tokens for previously authenticated clients if the server policy changes.

4 Implementation and Evaluation

Since IBM's TE policy engine in TrustBuilder already relies on X.509 certificates, it makes sense for the authorization server to issue an X.509 certificate as the trust token given to a newly authorized client. This has the advan-



(a) Example purchase request and response

(b) Authorization service architecture

Figure 2. Stock purchase scenario

tageous side effect that the creation of the X.509 certificate involves the generation of a private key for the web service client that its handler can use to sign the purchase messages that it sends to the stock exchange, thereby guaranteeing message integrity. Further, the token can be transmitted in the clear without fear of use by unauthorized parties, because only the stock purchase client's handler can prove ownership of the private key. Other reasonable options for a token include a Kerberos ticket, a signed SAML assertion, or a WS-Security token that can be issued and checked in accordance with WS-Trust.

Stock purchase messages and responses are sent over an unencrypted channel. A digital signature on a stock purchase request/response, as defined in the SOAP-dsig [19] or XMLDSIG [21] standards, will allow the recipient of a message to determine whether the message has been tampered with. The use of a secure channel (SSL/TLS) for the purchase and response messages would provide purchase confidentiality, but would hide the interaction from third-party auditors.[1] In our implementation, we chose to send purchase and confirmation messages in plaintext, to allow third party tracking of purchases. To guarantee integrity of purchase requests, they are signed. To keep the prototype simple, we do not guarantee message integrity on purchase responses; however, we could have designed it to do so in much the same way as the client signs its requests.

An attacker might try to replay a signed stock purchase message or response. Such an attack could trick the service into executing the same purchase operation multiple times. To address this, we have the stock purchase client add a sequence number to the security header of the SOAP message, and the stock purchase server remembers the highest sequence number of a message that has been sent with each received token. During a replay attack, the stock purchase server will detect a duplicate sequence number and can reject the message. (If the attacker modifies the sequence number, the message signature will not match and the server can still reject the message.) To prevent an overload of state space information at the stock purchase server, the stock purchase server sets a limit on the number of times a valid token can be used, and requires the client to renegotiate with the authorization service to receive a new token when the maximum number is reached.

Finally, an attacker might gain control of a machine between the client and the server. The digital signature on purchase messages and responses prevents message modification, but an attacker could also hold a request indefinitely, then send it on when it might have unintended consequences. For example, an attacker might intercept a purchase order and hold it until the price of the stock rises in order to raise more funds for the company, or intercept a sell order and wait until the price of the stock falls in order to purchase those shares himself at a better price. To prevent this, we add an expiration timestamp to the message. This will not guarantee that the message arrives in time, so the client will need contingency plans in this case, just as though the network were unavailable. The timestamp will, however, prevent any future damage. Similar problems can occur in the non-digital world, which is why stock purchase/sale requests often have a price limit or an expiration time built into the request.

To determine how much overhead trust negotiation adds

[1] The cipher negotiation in SSL/TLS does allow integrity protection without encryption, and is likely to be faster than XMLDSIG-based integrity. Unfortunately, this will not reduce the running time of the trust-negotiation phase, which is the bottleneck in our scenario. It would also require integration between the application layer (the TrustBuilder ticket) and the transport layer (SSL).

to a typical system, we measured the performance of a web service that simulates stock purchases. The web service takes two parameters (a text string representing the company ticker symbol, and an integer representing the number of shares) and returns a text string indicating whether the transaction succeeded. Thus the experiments essentially show the time spent in the authorization step, as they omit the time that would be required to actually purchase a stock, sign the purchase response, or set up a secure channel for the purchase. Every request to the web service and authorization server is for a stock purchase, and every request is eventually authorized.

To determine the effect of client and server policy complexity on authorization time, we varied the number of rounds in the negotiation. In particular, we modified the policies to require between one and ten iterations of the negotiation before trust was established. In each iteration, the client or server discloses a single credential. For its effect on performance, this is the worst-case scenario, as it maximizes the number of message rounds per credential disclosure. In the real world, we expect most negotiations to involve only one or two rounds of disclosures. However, it is important to understand the system's behavior with additional rounds of disclosures, as that is one potential avenue of attack.

The experiments ran on a Pentium 4 with 1 GB RAM running at 2.8GHz with Windows XP, the Apache Tomcat Server v5.0 with JDK/JWSDP v1.5, and TrustBuilder. The client and the server connected to each other through Java sockets. To eliminate network transmission times from the experiments, we ran both the client and the server on the same machine, although they connected to one another in the same manner as if they were on separate machines. To eliminate any caching effects, the client and server were both restarted after each purchase request and response (except for the double-purchase runs described below).

The experiments measure the total elapsed time from the initial purchase request creation at the client to the receipt of the confirmation at the client, for each level of policy complexity. To show the cost associated with using a secure channel to provide credential and policy confidentiality, the experiments include measurements with and without an SSL/TLS channel to the authorization server. The experiments with the secure channel include the time to set up the channel for each purchase request.

The experiments evaluate the cost of a single purchase request, including the authorization step, and also the cost of doing two purchase requests, where the second purchase request reuses the trust token that was generated for the first request. More precisely, during the second purchase request, the client's handler finds that it already has a token. It attaches the token to the client's request without contacting the authorization service, and forwards the message with token to the stock purchase web service. The web service's handler verifies the token and then passes the message to the stock purchase service.

Figure 3 shows the results of the experiments. Overall, the cost of obtaining a token using trust negotiation is high. The simplest form of trust negotiation (one client credential disclosed) takes 7 seconds, and each additional round of negotiation, with one additional disclosed credential, adds .5 seconds. The figure shows that the overhead for using a secure channel to communicate with the authorization server is noticeable, but still small (always less than .5 second) compared to the overall run time. When the client does two purchases in a row, the cost of attaching the token to the second request and having it verified is quite reasonable (less than .25 second).

Overall, the performance of TrustBuilder with TE is inadequate: 7 seconds is too long for a simple authorization request. We believe that this cost needs to be lowered, even though the following factors would tend to improve performance in real-world applications: (1) The high cost of obtaining a token would be amortized across many subsequent uses of the token in real life. (2) We would also expect to see benefits from caching of the stock purchase server's policies and credentials at the authorization server, which the experiments explicitly excluded across separate runs. (3) The policies in the experiments were designed to prolong the negotiation as much as possible—sending multiple credentials in one negotiation round will decrease the latency. As no researchers have addressed scalability issues for trust negotiation to date, there is surely much more that can be done to improve performance, compared to today's prototypes for policy evaluation and credential verification.

Beyond performance concerns, we found two other issues that should be addressed before trust negotiation can be deployed in critical applications. First, the ease of use of the technology needs improvement. We found that the process of writing TPL policies for the IBM TE engine was quite awkward, and demanded most of the coding time. Policy languages are an area of active research, and we encourage researchers to address language usability issues. We also found the tools for assembling and installing application scenario policies and credentials to be fragile and hard to use, with the result that the experiments took much longer than expected to set up. We encourage researchers to consider these aspects as well when creating policy management tools.

Second, we had concerns about the internal workings of TE. As TE is a closed system, we were unable to see how TE addresses such performance-, correctness-, and DoS-related issues as caching previously verified certificates, checking certificate chains, and requesting certificate revocation lists. We concluded that it is preferable to be able to examine the source code for a policy evaluation engine that

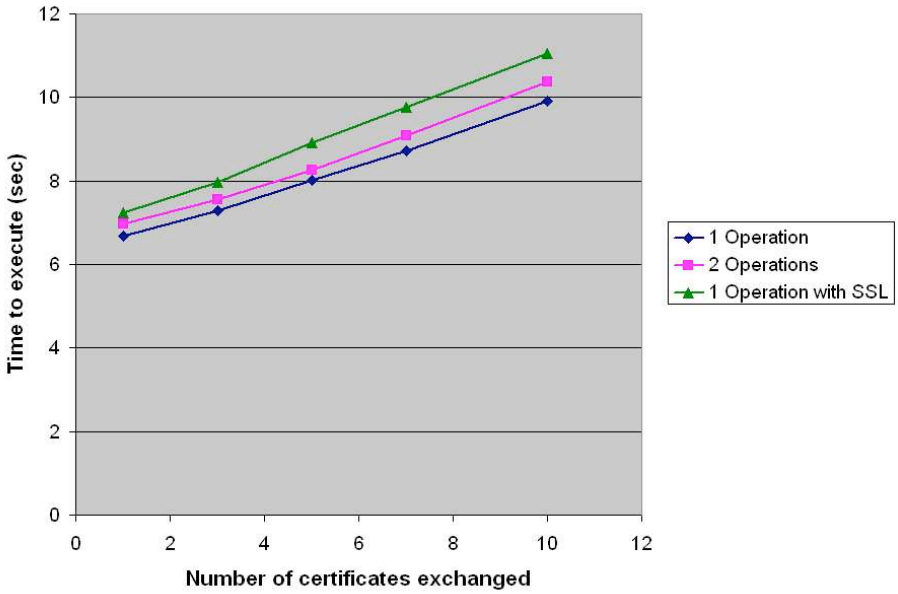


Figure 3. Performance results for the stock purchase scenario

is to be used in an open-systems authorization service.

The token-based approach may present opportunities for DoS attacks. One potential area of concern is the amount of time required to verify a trust token. Assuming that the authorization service's public key is already known, the stock purchase web service will still need to verify the stock client's trust token with the authorization service's public key, check the certificate revocation list of the authorization service (if one exists), verify the XMLDSIG signature of the stock purchase message, and check the sequence number and message expiration of the stock purchase message. As any third-party authorization service will require many or all of these steps, it will be important to minimize their cost, especially for invalid purchase requests. While the cost of these actions was reasonable in our experiments (less than .25 second), the cost could add up quickly if many clients attempted to buy stocks simultaneously.

Another opportunity for a DoS attack lies in the state information (the last sequence number) that the stock purchase service keeps for each valid trust token. State information is only kept for valid tokens, so a denial-of-service attack that tried to flood the server with additional valid tokens could be traced back to the attacker by logging the credentials the attacker presented at the authorization server. The amount of state information could be tuned dynamically by adding a temporal limit on the validity period of newly issued trust tokens; if the stock purchase server is overloaded with state information, it could ask the authorization service to shorten the validity period for newly issued tokens.

Our brokerage firms are responsible for issuing, updating, and revoking the certificates that identify their employees. Compared to traditional centralized distributed systems authorization, our reliance on third parties presents more entry points for a would-be attacker: if an attacker can obtain an employee certificate from a brokerage firm, then the attacker might be able to purchase or sell stock. An analysis of how to apply usage policies, both to certificate parameters such as key strength and expiration and to security practices by the third parties, is necessary before such a system can be deployed in practice.

5 Related Work

Many recent works have targeted trust management for distributed systems. We survey a number of them here, concentrating on those with well-developed runtime systems.

The KeyNote trust management system [1] allows a wide variety of policies, but does not support general-purpose credentials. KeyNote credentials are intended for use in obtaining access to a particular resource, and cannot in general be used for other purposes (unlike a driver's license, which can be used for many purposes besides proving eligibility to drive).

As discussed earlier, the Shibboleth project [18] provides an architecture for distributed attribute-based authorization that provides protection for sensitive attributes. The key differences between Shibboleth and a trust-negotiation-based approach are that Shibboleth does not support bidirectional

trust establishment or policies that involve client attributes attested to by multiple organizations (such as being covered by a particular insurance company *and* being a patient at a particular hospital), and requires organizations to join Shibboleth consortiums before their members' attributes can be used in authorization decisions.

The Traust project [13] also uses the TrustBuilder system as the basis of a third-party authorization server, which can be deployed to protect legacy resources. The Traust protocol requires a secure TLS connection and thus pushes security considerations such as integrity and availability to the transport layer. Traust was not targeted at a web services environment, and therefore does not have our handler-based architecture. Our performance analysis should also apply to a Traust service that issues X.509 credentials as trust tokens.

Koshutanski and Massacci [11] present algorithms for web service access control using digital credentials, along with reasoning about policy satisfiability in a stateless environment. They support non-monotonic policies for separation of duty and describe how to report some minimal set of credentials for the client to disclose or revoke to gain access. This work discusses the theory for discovering these minimal sets more than system issues such as third-party authorization, legacy application support, and prevention of attacks; however, it seems reasonable that their approach could be used as the basis of a prototype similar to ours.

Trust-X [3] offers a language and XML format for expressing trust statements, policies, and trust tickets. A trust ticket can be presented to the trust negotiation system to simplify future negotiations, and is not intended for use in third-party authentication—in particular, protection against replay attacks or interception of the trust ticket would be needed. As mentioned earlier, Trust-X could be used as the basis of a prototype similar to ours, and the same research group has also investigated the use of access control rules with web services [4].

The Cassandra project [2] has focused on the design of a Datalog-based policy language for real-world situations. The Cassandra language offers elegant role-related constructs inspired by its use in encoding policies for medical records disclosure in Britain. At run time, two parties cooperate by following the same algorithm to establish trust, which limits autonomy and opens up potential attack avenues. Cassandra's runtime system is not as advanced as its theoretical foundations (*e.g.*, it does not yet support verifiable, unforgeable credentials), but with a more mature runtime system it too could be used as the basis of an authorization system for web services.

Ziebermayr and Probst [23] describe how to apply access control rules to a web service and report on an implementation using a servlet filter to determine when access is granted. This is similar to our approach that uses handlers, but they assume a closed system with identity-based access control.

Like our project, .TRUST [12] aims to build a third-party token-based authorization system for web services that gives the client control over its own credentials. .TRUST uses distributed identity (a username/password combination) rather than attribute-based policies. .TRUST differs from our effort in that our authorization facilities are for an *open* system, which entails bilateral trust establishment and privacy protection for both parties.

The *idemix* [9] system offers a runtime system that supports anonymous credentials, i.e., credentials that one can use to prove that one has a particular attribute, without revealing one's identity. This runtime facility is backed up by strong theoretical guarantees of anonymity and correctness, including an approach to detecting collusion between parties. We did not consider the use of *idemix* for our facility, as stock purchases should not be anonymous; however, *idemix* would work well as the basis for a web services authorization facility where anonymity was desired.

6 Conclusion

Trust negotiation is a natural fit for an authorization service for a web service. A trust-negotiation-based authorization service can give trust tokens to newly authorized clients, which the clients present to gain access to a particular web service. The trust tokens can take a variety of forms (SAML assertions, Kerberos tickets) as well as the X.509 certificates used in our prototype implementation.

In the long run, trust negotiation has the potential to satisfy all the desiderata that we presented for web-services authorization facilities. Currently, however, our experiments showed that the runtime costs associated with the use of trust tokens are modest, but the cost to initially obtain a token is high (7 seconds minimum). Research is needed to address scalability considerations in trust negotiation policy engines and credential verification, as well as to address the difficulty of policy management.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The KeyNote Trust Management System," IETF RFC 2704, September 1999.

[2] M. Becker and P. Sewell, "Cassandra: Distributed Access Control Policies with Tunable Expressiveness," *IEEE Workshop on Policies for Distributed Systems and Networks*, June 2004.

[3] E. Bertino, E. Ferrari, and A. Squicciarini, "Trust-X: A Peer-to-Peer Framework for Trust Establishment," *IEEE TKDE*, July 2004, pp. 827-842.

[4] R. Bhatti, E. Bertino, and A. Ghafoor, "A Trust-based Context-Aware Access Control Model for Web-Services," *IEEE Intl. Conf. on Web Services*, June 2004.

[5] E. Damiani, S. di Vimercati, and P. Samarati, "Towards Securing XML Web Services," *ACM Workshop on XML Security*, November 2002.

[6] C. Ellison, B. Franz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "RFC 2693: SPKI Certificate Theory," http://www.faqs.org/frcs/rfc2693.html.

[7] C. Gunter and T. Jim, "Policy-Directed Certificate Retrieval," Software Practice and Experience, 2000.

[8] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith, "Advanced Client/Server Authentication in TLS," *Network and Dist. System Security Symp.*, February 2002.

[9] "*idemix*: pseudonymity for e-Transactions," http://www.zurich.ibm.com/security/idemix.

[10] T. Jim, "SD3: A Trust Management System with Certified Evaluation," *IEEE Symp. on Security and Privacy*, May 2001.

[11] H. Koshutanski and F. Massacci, "Interactive Credential Negotiation for Stateful Business Processes," *Proc. Intl. Conf. on Trust Management (iTrust)*, May 2005.

[12] H. Kung, F. Zhu, and M. Iansiti, "A Stateless Network Architecture for Inter-enterprise Authentication, Authorization and Accounting," *Intl. Conf. on Web Services*, June 2003.

[13] A. Lee, *Traust: A Trust Negotiation Based Authorization Service for Open Systems*, M.S. thesis, University of Illinois at Urbana-Champaign, August 2005.

[14] M. Lehmann, "Developer Web Services: Creating Web Services, Part 2," *Oracle Magazine*, March/April 2004.

[15] J. Li, N. Li, and W. Winsborough, "Automated Trust Negotiation Using Cryptographic Credentials," *ACM Conf. on Computer and Communications Security*, November 2005.

[16] W. Nejdl, D. Olmedilla, and M. Winslett, "PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web," *Secure Data Management Workshop*, 118-132, 2004.

[17] T. Ryutov, L. Zhou, C. Neuman, T. Leithead, and K. E. Seamons, "Adaptive Trust Negotiation and Access Control," *ACM SACMAT*, June 2005.

[18] *Shibboleth Architecture Technical Overview*, Internet2 Working Draft, 8 June 2005, http://shibboleth.internet2.edu/docs/draft-mace-shibboleth-tech-overview-latest.pdf.

[19] *SOAP Security Extensions: Digital Signature*, W3C Note, 6 February 2001, http://www.w3.org/TR/2001/NOTE-SOAP-dsig-20010206/.

[20] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu, "The TrustBuilder Architecture for Trust Negotiation," *IEEE Internet Computing*, volume 6, number 6, November/December 2002, pages 30-37.

[21] *XML-Signature Syntax and Processing*, W3C Recommendation, 12 February 2002, http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[22] T. Yu, M. Winslett, and K. E. Seamons, "Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation," *ACM Transactions on Information Systems Security*, 6(1): 1-42, 2003.

[23] T. Ziebermayr and S. Probst, "Web Service Authorization Framework," *Proc. Intl. Conf. on Web Services*, June 2004.