Towards Standards-Compliant Trust Negotiation for Web Services (Extended Version)*

Adam J. Lee and Marianne Winslett
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
{adamlee, winslett}@cs.uiuc.edu

Abstract

Web services are a powerful distributed computing abstraction in that they enable users to develop workflows that incorporate data and information processing services located in multiple organizational domains. Fully realizing the potential of this computing paradigm requires a flexible authorization mechanism that can function correctly without a priori knowledge of the users in the system. Trust negotiation has been proposed as a viable solution to this problem, but doing so within the framework provided by existing web services standards remains an unsolved problem. In this paper, we show how existing web services standards can be extended to enable fully standards-compliant support for trust negotiation. We also show that it is possible to compile trust negotiation policies specified using the WS-SecurityPolicy standard into a representation that is suitable for analysis by Clouseau, a highly-efficient trust negotiation policy compliance checker. Lastly, we show that the TrustBuilder2 framework for trust negotiation can be parameterized to act as a trust engine that can be used by the WS-Trust standard to facilitate these negotiations.

1 Introduction

Web services and other service oriented architectures stand poised to usher in a new era of distributed computing. Standards such as WSDL [8] and UDDI [20] enable entities to describe and deploy computational services that can be searched for, discovered, and utilized by other entities. Furthermore, languages such as BPEL4WS [7] can be used to describe potentially complex workflows that utilize data and computational services spread across multiple administrative domains. Fully realizing the potential of this computing paradigm requires a flexible authorization mechanism that can function correctly without a priori knowledge of the users in the system, as this would allow for the discovery and composition of new services at runtime. Unfortunately, existing web services security mechanisms are largely identity-based; this requires that a user be, in some sense, "hard-wired" into every administrative domain that they wish to access services from. This severely hinders the full potential of the web services model.

Trust negotiation [21] has previously been proposed as an appropriate authorization model for use in a web services context. In trust negotiation, resources are protected by

*A shorter version of this paper appears as [14].

attribute-based access policies. Entities use digital credentials issued by third-party attribute certifiers (e.g., professional organizations, employers, government bodies, etc.) to prove various characteristics about themselves and their surrounding environment. Because these attributes might also be considered sensitive, they can optionally be protected by release policies constraining the individuals to whom they can be disclosed. As such, a trust negotiation session evolves into a bilateral and iterative exchange of policies and credentials with the end goal of developing new trust relationships on-the-fly. Because these types of systems allow resource administrators to specify the *intension* of a policy, rather than its logical *extension* (i.e., an explicit access control list), authorized entities can gain access to available resources without requiring that their identity be known a priori.

The decentralized and expressive nature of trust negotiation makes it a natural fit for web services computing environments. As a result, previous research has explored this connection to some extent [4, 10]. While this work has made important contributions to the fields of trust negotiation and authorization architectures for web services, it has not addressed one important consideration: compliance with existing web services standards. There are currently a myriad of security-oriented standards in the web services domain that aim to enable many advanced security features. Rather than further cluttering this space with yet other standards, it is important to consider how existing standards might be used to support more advanced authorization paradigms, such as trust negotiation. In this paper, we consider exactly this problem. Specifically, we make the following contributions:

- We show how the existing token-based security model described by the WS-SecurityPolicy [16] standard can be used to specify trust negotiation policies. We then describe a standards-compliant *claims dialect* that can be used in conjunction with WS-SecurityPolicy to enable the specification of more expressive authorization policies.
- We propose extensions to WS-Trust's challenge/response framework [17] that can be used to facilitate trust negotiation sessions in a fully standards-compliant manner. These extensions do not limit the strategies, policies, or credential types that can be used during the trust negotiation process.
- We present a procedure for compiling trust negotiation policies specified using the WS-SecurityPolicy standard into a format suitable for analysis by Clouseau [13], a highly efficient and language-agnostic policy compliance checker for trust negotiation systems.
- We show that the TrustBuilder2 framework for trust negotiation can be parameterized to act as a trust engine—as defined by WS-Trust—that is capable of driving standards-compliant trust negotiation sessions.

The rest of this paper is organized as follows. In Section 2 we discuss related efforts in using trust negotiation within a web services context, as well as overview important web services security and trust standards. Section 3 describes how trust negotiation policies can be specified using the WS-SecurityPolicy standard. In Section 4, we show how to execute trust negotiations through extensions to the WS-Trust standard. Section 5 focuses on systems issues, including a correct and complete compilation procedure that enables policies specified using WS-SecurityPolicy to be translated into a format suitable for analysis by the Clouseau compliance checker. We further describe how the TrustBuilder2 framework for trust negotiation can be parameterized to function as the trust engine used by WS-Trust during these negotiations. We then present our conclusions in Section 6.

2 Related Work

In this section, we provide background information on a number of relevant web services security standards, as well as discuss related work involving the use of trust negotiation in

the web services domain.

2.1 Web Services Security Standards

At their most basic level, web services are nothing more than software components that communicate with one another by sending XML messages enclosed in SOAP envelopes. Each of these envelopes consists of a header containing routing information and other meta-data, as well as a body that encapsulates the "payload" of the message. Since these messages are often routed over public networks, such as the Internet, they are susceptible to observation and tampering by unauthorized entities. The WS-Security standard [18] defines a number of useful primitives that can help protect against these types of threats. This standard defines an optional security header that can be used to transport key material, message authentication codes, and various types of security tokens that can be used to authenticate users or protect the confidentiality or integrity of messages.

In order to take full advantage for the security features enabled by WS-Security, service administrators need some means of defining the security requirements for a web service. The WS-SecurityPolicy standard [18] defines *policy assertions* that allow administrators to place constraints on the types of authentication tokens that need to be presented to gain access to a service, the portions of incoming and outgoing messages that need to be encrypted or authenticated, suites of cryptographic algorithms that are supported, and other security-relevant properties of their service. The basic policy structures and connectives defined by WS-Policy [19] are then used to combine these policy assertions into comprehensive *security policies*.

The final web services standard that we will leverage in this paper is WS-Trust [17]. Properly exchanging and using the types of security tokens defined in the WS-Security standard requires that each party involved can assess the trustworthiness of each security token that it acquires. WS-Trust leverages the security primitives defined in WS-Security along with additional extensions to enable services to carry out protocols designed to issue, renew, and validate security tokens, as well as broker trust relationships. In Section 4, we will describe how the negotiation and challenge extensions to WS-Trust can be used to carry out trust negotiation sessions in a standards-compliant manner.

2.2 Trust Negotiation for Web Services

While much research effort has been placed into the foundations of trust negotiation—such languages for expressing resource access policies (e.g., [2, 3, 9, 15]), protocols and strategies for conducting trust negotiations (e.g., [4, 11, 12, 23]), and logics for reasoning about the outcomes of these negotiations (e.g., [6, 22])—only a few research groups have investigated the applications of trust negotiation within the web services domain.

Bertino et al. describe Trust-$\mathcal{X}$ [4], an XML-based framework for supporting trust negotiations in peer-to-peer systems. In Trust-$\mathcal{X}$, each user creates an $\mathcal{X}$-profile that stores $\mathcal{X}$-TNL certificates [3] describing their attributes, uncertified declarations containing information about the user (e.g., preferences, phone numbers, or other such information), and $\mathcal{X}$-TNL policies to protect their sensitive resources. Since these data are all specified using XML, they can be queried or constrained using standard query languages, such as XQuery [5]. To allow users to optimize various aspects of the trust negotiation process, Trust-$\mathcal{X}$ supports a variety of interchangeable trust negotiation strategies. Another particularly innovative feature of the Trust-$\mathcal{X}$ framework is its support for *trust tickets*, which are receipts that attest to the fact that a user recently completed some negotiation with another party that be presented within some limited lifetime to bypass redundant portions of future negotiations. Although Trust-$\mathcal{X}$ makes heavy use of XML, it was not designed specifically for the web services environment. In particular, the authors do not specify how these trust negotiations might be carried out within the framework provided by other web services protocols and standards.

In [10], Koshutanski and Massacci describe a trust negotiation framework designed for web services. This framework facilitates the composition of access policies across the constituent pieces of a workflow, the discovery of credentials needed to satisfy these policies, the management of the distributed access control process, and the logic to determine what missing credentials must be located and provided to satisfy a given policy. This work operates at the business process level by determining and satisfying the composite access control policy for a workflow prior to its execution; as a result, existing web services security standards are not used. Furthermore, policies are represented using a datalog-based language, rather than an existing standards-compliant language.

3 Specifying Trust Negotiation Policies

During a trust negotiation session, attribute-based policies are used to describe the characteristics of the entities authorized to access a given resource. Digital credentials are then used to satisfy these policies; in a web services context, these credentials can be represented using the formats specified in WS-Security and its extensions. In this section, we address the problem of representing trust negotiation *policies* for web services. We first show that policy assertions defined in WS-SecurityPolicy can be used to specify basic trust negotiation policies, and then present a standards-compliant claims dialect that extends WS-SecurityPolicy to enable the specification of more expressive trust negotiation policies.

3.1 Basic Policy Specification

WS-SecurityPolicy takes a token-based approach to security, in that policies identify specific *security tokens* that must be presented in order to gain access to a particular service. The WS-SecurityPolicy specification defines policy assertions that can be used to require the use of Kerberos tickets, SAML assertions, and X.509 certificates, as well as other security token formats. As an example, the following policy assertion requires the use of an X.509 certificate issued by the Better Business Bureau's (fictitious) security token service:

```
<sp:X509Token xmlns:sp="..." xmlns:wsa="...">
  <sp:IssuerName>C=US/O=Better Business Bureau/CN=sts.bbb.org</sp:IssuerName>
</sp:X509Token>
```

Trust negotiation policies are typically more complicated than this, however, as they can include *multiple* attribute constraints. Requiring the use of multiple security tokens can be accomplished through the use of the basic policy connectives defined by WS-Policy. WS-Policy defines the `ExactlyOne` and `All` connectives, which require that either *one* or *all* subclauses of a particular clause in a policy be satisfied in order for that clause of the policy to be satisfied. Although these two connectives can be used to express any arbitrary policy structure, the WS-Policy specification recommends that policies be expressed in disjunctive normal form (DNF) using a policy structure of the following form:

```
<wsp:Policy xmlns:wsp="...">
  <wsp:ExactlyOne>
    <wsp:All>
      <!-- Policy assertion (1,1) -->
      ...
      <!-- Policy assertion (1,n_1) -->
    </wsp:All>
    ...
    <wsp:All>
      <!-- Policy assertion (m,1) -->
      ...
      <!-- Policy assertion (m,n_m) -->
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

```
<wsp:Policy xmlns:wsp="..." xmlns:sp="...">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:X509Token xmlns:wsa="...">
        <sp:IssuerName>
          C=US/O=State University/OU=Registrar/CN=sts-reg.stateu.edu
        </sp:IssuerName>
      </sp:X509Token>
      <sp:X509Token xmlns:wsa="...">
        <sp:IssuerName>
          C=US/O=ACM/CN=sts.acm.org
        </sp:IssuerName>
      </sp:X509Token>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 1: An example trust negotiation policy requiring users to present X.509 certificates from the State University registrar, as well as the ACM.

The `ExactlyOne` clause in the above policy indicates that exactly one of its child nodes must be satisfied in order for the policy to be satisfied. Since each child node is an `All` clause, satisfying this policy requires that *all* policy assertions (`i`, `j`) are satisfied for some `i` where $1 \leq \texttt{i} \leq \texttt{m}$ and for all `j` such that $1 \leq \texttt{j} \leq \texttt{n\_i}$.

Combining the security token policy assertions from WS-SecurityPolicy with the policy connectives defined in WS-Policy allows us to specify a range of interesting trust negotiation policies. For example, consider a service that wishes to be protected by a policy requiring that users present X.509 certificates issued by the registrar of State University and the ACM; this would indicate that authorized users of the service need to be students of State University and members of the ACM. Assuming that State University's registrar and the ACM each run an online security token service (STS) that manages the credentials issued within their respective domains, Figure 1 illustrates how such a policy could be written in a standards-compliant manner.

3.2 Encoding Advanced Attribute Constraints

While the above, strictly token-based approach to trust negotiation policy specification works in some circumstances, it is inadequate for others. For example, consider complex credentials such as driver's licenses that contain information about the type of vehicles the bearer is authorized to drive and the date of birth of the bearer, or employee IDs that indicate the employee's rank, department, and year of hire. The policies used in the previous section can only determine whether an entity has an employee ID or driver's license, but cannot constrain the attribute fields—also known as claims—encoded in the certificate.

The authors of the WS-SecurityPolicy and WS-Trust standards recognize that placing constraints on the claims encoded in a security token is an important aspect of security policy specification. As such, these standards define an optional `Claims` element that can be included in the security token policy assertions that make up a given security policy. These standards do not specify the contents of given `Claims` element; to allow for maximum extensibility, third parties can define claims *dialects* that specify the format and contents of these elements.

To facilitate the use of more expressive—yet standards-compliant—trust negotiation policies within the WS-SecurityPolicy framework, we have developed one

Element	Description
/cl:Claim	This element is used to encode a constraint on some claim encoded in the security token to which it refers. These constraints take the form of (attribute, operation, value) triples.
/cl:Claim/cl:Attribute	The name of the attribute or claim to which this constraint refers.
/cl:Claim/cl:Op	The operation portion of a constraint triple. Acceptable values for this field are `EQ`, `GT`, `LT`, `GTEQ`, and `LTEQ`. These values denote "equals," "greater than," "less than," "greater than or equal to," and "less than or equal to," respectively.
/cl:Claim/cl:Value	The value field of the constraint triple.
/cl:Ownership	This element is used to indicate whether proof of ownership of the security token to which it refers needs to be demonstrated when the token is disclosed.
/cl:Ownership/@Status	This optional attribute may be set to either `true` or `false` depending on whether proof of ownership is required. If this attribute is not present, a default value of `true` is assumed.

Table 1: Descriptions of the elements making up our claims dialect.

such claims dialect. Our claims dialect allows policy writers to place an arbitrary number of (attribute name, comparison operator, value) constraint triples on the claims encoded in a security token. This format was chosen because it is sufficiently expressive to represent instances of the constraint checking problem. For example, the constraint triple (License Type, EQ, CDL) would require that the "License Type" field of a particular driver's license security token be set to the value "CDL." Furthermore, our claims dialect provides a mechanism through which policy writers can require not only the disclosure of a particular security token, but also a demonstration of proof-of-ownership. This enables explicit differentiate between credentials that must be *owned* by the individual requesting access to a particular service and other supporting credentials that must be presented. The XML elements defined by this claims dialect are summarized in Table 1; a more detailed treatment of this claims dialect can be found in the XML schema defining the dialect (see Appendix A).

Figure 2 contains a more complex version of the policy presented in Figure 1. This version of the policy leverages our claims dialect to restrict service access to *graduate* students of State University who have been members of the ACM *since at least 2006*. The use of the `Ownership` element inside each of the `Claims` elements requires that proof of ownership be demonstrated for both tokens.

4 Trust Negotiation Using WS-Trust

Now that we have described how trust negotiation policies can be specified in a standards-compliant manner, we must show how trust negotiation *protocols* can be executed within the framework provided by existing web services standards.

4.1 WS-Trust Basics

As described in Section 2, the WS-Trust standard focuses on the brokerage of trust relationships between entities in a web services environment. In the trust model articulated in the WS-Trust standard, trust relationships are represented as security tokens. For example, if Alice runs a web service that she would like to allow Bob's friends to use, she would protect her web service with a WS-SecurityPolicy policy requiring a security token issued by Bob. This type of token would serve as formal

```
<wsp:Policy xmlns:wsp="..." xmlns:sp="...">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:X509Token xmlns:wsa="...">
        <sp:IssuerName>
          C=US/O=State University/OU=Registrar/CN=sts-reg.stateu.edu
        </sp:IssuerName>
        <wst:Claims Dialect="http://dais.cs.uiuc.edu/claim.xsd">
          <cl:Claim>
            <cl:Attribute>Type</cl:Attribute>
            <cl:Op>EQ</cl:Op>
            <cl:Value>Graduate Student</cl:Value>
          </cl:Claim>
          <cl:Ownership Status="true"/>
        </wst:Claims>
      </sp:X509Token>
      <sp:X509Token xmlns:wsa="...">
        <sp:IssuerName>
          C=US/O=ACM/CN=sts.acm.org
        </sp:IssuerName>
        <wst:Claims Dialect="http://dais.cs.uiuc.edu/claim.xsd">
          <cl:Claim>
            <cl:Attribute>MemberSince</cl:Attribute>
            <cl:Op>LTEQ</cl:Op>
            <cl:Value>2006</cl:Value>
          </cl:Claim>
          <cl:Ownership Status="true"/>
        </wst:Claims>
      </sp:X509Token>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 2: A more complex example trust negotiation policy that makes use of our claim dialect.

proof of the fact that Bob is friends with a certain individual. The WS-Trust standard then goes on to define the protocols that can be used to issue, renew, revoke, and check the validity of security tokens; later in this section, we will show that the token issuance protocol described by WS-Trust can be extended to enable native support for trust negotiation. As a means of introduction to this protocol, we now discuss an example execution of the basic protocol.

Figure 3 illustrates how the scenario described in our previous example might make use of the WS-Trust standard to control access to Alice's web service. When a user Charlie tries to access Alice's service, he is returned a WS-SecurityPolicy policy indicating that he must present a security token that identifies him as a friend of Bob's before he can access the service in question. Charlie does not have such a token, so he contacts Bob's STS and sends a SOAP message containing a `RequestSecurityToken` element indicating that he would like to be issued a security token identifying him as a friend of Bob's. This message includes a copy of Charlie's public key certificate—which is attached as described in the WS-Security standard—and is digitally signed to ensure its authenticity. Bob's STS then checks to see if Charlie is on the access control list (ACL) containing the names of Bob's friends. Since Charlie is on this list, the STS generates a security token for Charlie, embeds the token in a `RequestedSecurityTokenResponse` element, and includes this element in the body of a SOAP message. This message is then signed, encrypted, and returned to Charlie, who can then use the encapsulated token to access Alice's service.

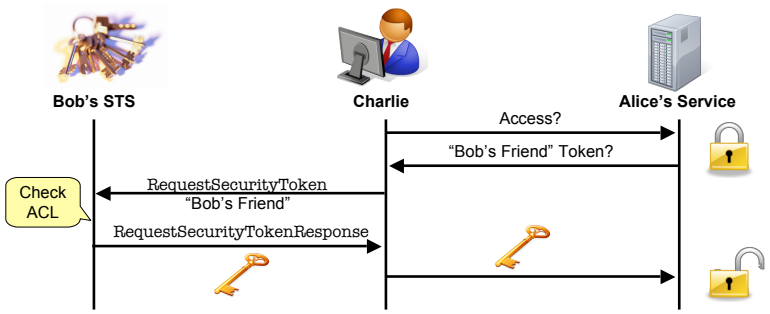

Figure 3: An example of the WS-Trust token issuance mechanism.

4.2 Trust Negotiation Extensions to WS-Trust

Since the issuance of a security token might not always fit within a single request and response, WS-Trust includes a *negotiation and challenge framework* that enables support for more complex token issue protocols. After a requestor discloses an initial `RequestSecurityToken` message, this framework allows the requestor and STS to send any number of `RequestSecurityTokenResponse` messages containing arbitrary XML structures to one another before the final `RequestSecurityTokenResponse` message containing a new security token is disclosed (or a fault is generated). While these extensions were intended to support basic challenge/response protocols and legacy key exchange protocols, they can also be used to support trust negotiation.

Table 2 describes the `TNInit` and `TNExchange` XML elements that we have defined to encapsulate trust negotiation sessions within the WS-Trust negotiation and challenge framework. The `TNInit` element is exchanged by participants during the first round of the negotiation and contains information used to parameterize the negotiation that is about to take place. The remaining rounds of the negotiation consist of exchanges of `TNExchange` elements containing `PolicyCollection` and `TokenCollection` elements describing the policies and credentials being disclosed, respectively. Policies are encoded as described in Section 3, while security tokens are contained in `Token` elements that include a token type descriptor (in the form of a URI), the token itself (encoded as described in the WS-Trust specification), and an optional proof of ownership.

The above two-phase negotiation process enables support for an arbitrary array of trust negotiation protocols within the WS-Trust framework. Since the entities participating in the negotiation use the exchange of `TNInit` objects to choose which negotiation strategies and security token formats will be supported, the security token and policy exchanges that take place during the later phase of the negotiation can occur in accordance with these initial choices. Furthermore, the schema defining the `TNInit` and `TNExchange` elements (see Appendix B) can itself be easily extended to include support for the exchange of data items other than policies and security tokens (e.g., proof fragments [1, 22], uncertified claims [4, 6], or trust tickets [4]). Another benefit of this method of supporting trust negotiation is that the tokens issued by the STS function in many ways like the "trust tickets" described by Bertino et al. in [4]. That is, after a *single* successful trust negotiation, a service requestor can access the protected web service *many times* within the lifetime of the token issued by the STS.

Element	Description
tn:TNInit	This element is used to encapsulate initialization information that needs to be passed between negotiation parties.
tn:TNInit/tn:SignatureMaterial	This holds one party's contribution to the signature challenge used when proving token ownership.
tn:TNInit/tn:StrategyFamily	This element identifies one strategy family [23] supported by the sending entity. This element may occur multiple times in the first `TNInit` element, indicating that multiple strategy families are supported. The `TNInit` element returned by the second negotiation participant must include exactly one copy of this element, which indicates the strategy that was chosen for use during this negotiation.
tn:TNInit/tn:TokenFormat	This element identifies one security token type supported by the sending entity. This element may occur multiple times, indicating that multiple security token formats are supported.
tn:TNExchange	This element is used to encapsulate all information transferred during one exchange of a trust negotiation session.
tn:TNExchange/tn:TokenCollection	This element contains one or more `tn:Token` elements embodying the security tokens disclosed during a single trust negotiation exchange.
tn:TNExchange/tn:PolicyCollection	This element contains one or more `wsp:Policy` elements embodying the trust negotiation policies disclosed during a single trust negotiation exchange.
tn:Token	This element encapsulates information describing a single security token that is being disclosed.
tn:Token/wst:TokenType	This element contains a URI describing the type of security token being disclosed.
tn:Token/wst:RequestedSecurityToken	The security token being disclosed is encoded in this element, which is defined in the WS-Trust specification.
tn:Token/tn:OwnershipProof	This optional element contains a Base64-encoded representation of a response to a proof of ownership challenge for this security token.

Table 2: Descriptions of the elements making up our extensions to the WS-Trust negotiation and challenge mechanism.

4.3 An Example Standards-Compliant Trust Negotiation

In Figure 4, we see that Charlie is attempting to access a web service that uses trust negotiation authorization controls. Upon requesting access to the service, Charlie is told that he must present a security token issued by State University's STS in order to access the service. He contacts the STS and sends a SOAP message containing a `RequestSecurityToken` element indicating that he needs a security token to access the protected web service. The STS returns a SOAP message containing `RequestSecurityTokenResponse` element that initiates a trust negotiation with Charlie. This element contains a `TNInit` element containing initialization information for the trust negotiation session, as well as a `TNExchange` element containing a `PolicyCollection` element that includes the policy from Figure 2. Recall that this policy requires users to prove that they are graduate students at State University *and* that they have been a member of the ACM since at least 2006.

Upon receiving this message, Charlie creates a new SOAP message to the STS consisting of a `RequestSecurityTokenResponse` message containing a `TNInit` element to finalize the negotiation parameters, as well as a `TNExchange` element. The `TNExchange` contains a `TokenCollection` element that includes a copy of his ACM membership token (and its corresponding proof of ownership) and a `PolicyCollection` element containing a single `Policy` element requiring that the STS prove that it is certified by State University. This message is then sent to the STS, which returns another SOAP message to Charlie containing a

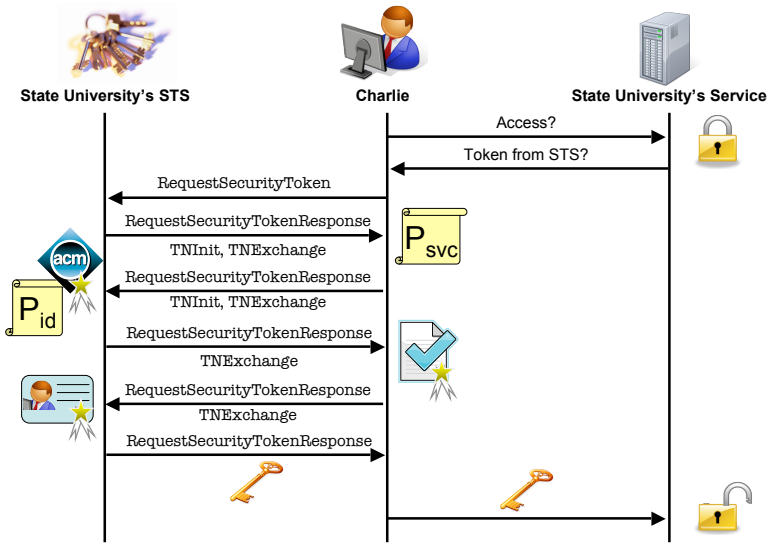

Figure 4: An example illustrating the use of extensions to the WS-Trust negotiation and challenge framework to facilitate trust negotiation sessions.

`RequestSecurityTokenResponse` with an embedded `TNExchange` element containing a `TokenCollection` element that includes the requested security token (and its corresponding proof of ownership). This message satisfies Charlie, who then discloses his Student ID token, which identifies him as a graduate student of State University. At this point, the STS returns a SOAP message containing the final `RequestSecurityTokenResponse` element, which includes a new security token indicating that Charlie has satisfied the web service's policy. Charlie then discloses this token to the web service and is granted access.

5 Systems Considerations

We now focus on the *systems* aspects of the trust negotiation process. We first show that policies expressed as in Section 3 can be compiled into a format suitable for analysis by CLOUSEAU, an efficient trust negotiation policy compliance checker. This not only reduces implementation overheads, but also establishes CLOUSEAU as a *general-purpose* compliance checker capable of analyzing both Datalog-based and industry-standard policy languages. We then show that the TrustBuilder2 framework for trust negotiation can be parameterized to function as a trust engine capable of driving these types of negotiation sessions.

5.1 Efficient Policy Compliance Checking

Given a policy p and a set of security tokens S, a *policy compliance checker* finds one or more minimal subsets of S that can be used to satisfy p. We call such minimal subsets *satisfying sets* of security tokens. Compliance checkers are used to find sets of local security tokens that can be disclosed to satisfy a remote policy, as well as to determine whether the security tokens disclosed by a remote party can be used

```
;; This policy is satisfied by graduate students at State University
;; who have been members of the ACM since at least 2006.
(defrule rule-service-access
  (credential (id ?istud) (issuer ?issstud) (owned true) (map ?mstud))
  (credential (id ?iacm) (issuer ?issacm) (owned true) (map ?macm))
  (test (eq ?issstud "C=US/O=State University/OU=Registrar/CN=sts-reg.stateu.edu"))
  (test (eq "Graduate Student" (?mstud get "Type")))
  (test (eq ?issacm "C=US/O=ACM/CN=sts.acm.org"))
  (test (<= (?macm get "MemberSince") 2006))
=>
  (assert (satisfaction (resource-name server)
          (credentials ?istud ?iacm))))
```

Figure 5: The policy presented in Figure 2 specified using CLOUSEAU's policy syntax.

to satisfy some local policy. Before the techniques outlined in this paper can be put to use, we require a compliance checker capable of analyzing policies written using the WS-Policy and WS-SecurityPolicy standards.

CLOUSEAU is an optimized policy compliance checker designed for trust negotiation systems [13]. Internally, CLOUSEAU represents policies as sets of *patterns* placing constraints on the collection of security tokens that must be presented to access a given resource. When invoked, CLOUSEAU translates the provided set of security tokens into an abstract *object* representation and then leverages efficient pattern matching algorithms to determine the collection of *all* satisfying sets of security tokens. Space limitations prevent a full discussion of the format of the constraint patterns analyzed by CLOUSEAU, so we instead refer interested readers to [13] for more details. As an introduction to CLOUSEAU's policy syntax, Figure 5 shows the policy presented in Figure 2 specified using this syntax.

In [13] the authors describe a compilation procedure for translating role-based policies written in the RT_0 and RT_1 policy languages [15] into the intermediate policy representation analyzed by CLOUSEAU. We now describe such a compilation procedure that can be used to translate policies specified as in Section 3 into a format suitable for analysis by CLOUSEAU. This translation is actually quite natural, as the token-based approach to trust and security embodied by WS-Trust maps directly onto the intermediate policy language used by CLOUSEAU.

In presenting the following compilation procedure, we assume that policies are expressed in DNF, as recommended by [19]. That is, we assume that policies are a collection of n `All` clauses, each identifying one satisfying set of security tokens for the policy. The ith such `All` clause in the policy should be processed as follows. First, a new rule will be created for this `All` clause:

```
(defrule rule-<i>
```

In the above rule, the `<i>` will be replaced with a counter indicating which `All` clause the rule represents. Assume this `All` clause has m `Token` elements. The kth such element will be processed as follows. First, a constraint will be added to the policy requiring that this token be presented:

```
(credential (id ?id-<k>) (issuer ?iss-<k>) (owned ?o-<k>) (map ?m-<k>))
```

If this `Token`'s `Claims` element or `IssuerName` element specifies that the token must be issued by some specific issuer, `<issuer>`, the following test will be added to `rule-<i>`:

```
(test (eq ?iss-<k> <issuer>))
```

If this `Token`'s `Claims` element contains the assertion `<Ownership Status="true">`, then the following test will be added to `rule-<i>`:

```
(test (eq ?o-<k> true))
```

For all other constraint triples encoded in the `Claims` element of this token, the following test will be inserted. Note that `<op>` is either `eq`, `<`, `>`, `<=`, or `>=` depending on whether the operation encoded in the constraint tuple is `EQ`, `LT`, `GT`, `LTEQ`, or `GTEQ`. Similarly, `<name>` and `<value>` are placeholders for the attribute name and constraint value identified in the constraint triple.

```
(test (<op> (?m-<k> get <name>) <value>))
```

After each `Token` element in the ith `All` clause has been processed as above, `rule-<i>` will be terminated as follows:

```
=>
  (assert (satisfaction (resource-name rule-<i>)
          (credentials ?id-1 ... ?id-<m>))))
```

This process then repeats for each other `All` clause defined by the policy. We now present the following theorem regarding the correctness and completeness of this compilation procedure:

Theorem 1. *Assume that a trust negotiation policy p specified using the WS-Policy and WS-SecurityPolicy specifications is compiled using the above procedure into a* Clouseau *policy p'. Given the policy p' and a set of security tokens S, the satisfying sets $s_1, \ldots, s_n$ returned by* Clouseau *are* exactly *the subsets of S that satisfy the original policy p.*

Proof. Recall that the policy p is expressed in DNF form; that is, p is a disjunction of n conjunctive clauses. Note that each such conjunctive clause identifies a unique set of security tokens that can be presented to satisfy p. For each conjunctive clause c_i containing identifying m_i security tokens, the compilation process defined above creates one Clouseau rule containing m_i patterns defined to match the tokens identified by c_i. Furthermore, for each security token t_j identified by c_i, the above process creates one `test` clause to check each claim constraint imposed on t_j by the policy p. Since no other rules are inserted into the policy p', Clouseau cannot find any satisfying sets other than those identified by p when analyzing the policy p'. Similarly, since one such rule is created for each conjunctive clause (i.e., satisfying set) in p, Clouseau finds *all* satisfying sets in p when invoked on the policy p'. □

5.2 TrustBuilder2 as a WS-Trust Trust Engine

The WS-Trust standard defines a *trust engine* as "a conceptual system that evaluates the security-related aspects of a message" [17]. Revisiting the basic WS-Trust token issuance example from Section 4.1, the trust engine would have been responsible for checking to see that Charlie was on Bob's list of friends. To execute the more complex example from Section 4.3, a more powerful trust engine would be required. This trust engine would need to determine which policies and/or security tokens should be disclosed at each round of the negotiation as a function of the existing negotiation state and the policies and/or security tokens that were received during the previous round. We now argue that in the future such a trust engine could—with minimal effort—be implemented using TrustBuilder2, an extensible open-source framework for trust negotiation.[1]

[1] TrustBuilder2 is available for download at `http://dais.cs.uiuc.edu/tn`.

To substantiate this claim, we must show that TrustBuilder2 can analyze policies specified using WS-Policy and WS-SecurityPolicy, and that it is at least as extensible as the trust negotiation extensions to WS-Trust described in Section 4. Recall from Section 5.1 that trust negotiation policies specified using WS-Policy and WS-SecurityPolicy can be compiled into a format that is analyzable by the CLOUSEAU compliance checker. Since TrustBuilder2 supports CLOUSEAU natively, it can analyze policies specified as described in Section 3. In Section 4.2, we showed that the extensibility afforded by our extensions to WS-Trust comes from two sources: the ability to support arbitrary trust negotiation strategies and security token formats, and the ability to extend the `TNExchange` element to transport trust negotiation evidence other than policies and security tokens.

The TrustBuilder2 framework makes use of an extensible data type hierarchy that users can extend to add support for new security token formats, policy languages, or trust negotiation evidence types (e.g., trust tickets, etc.). Additionally, the primary components of a trust negotiation system—including strategies—are represented as abstract interfaces that can be extended or replaced by users of the system. TrustBuilder2 also leverages a two-phase negotiation model in which participants first exchange InitBrick data structures allowing them to establish a mutually-acceptable system configuration. The remaining rounds of the negotiation involve the exchange TrustMessage objects that encapsulate the policies, security tokens, and other forms of evidence exchanged during the negotiation; note that this mirrors the exchange of `TNInit` and `TNExchange` elements described in Section 4.2. As a result, each message exchange during our trust negotiation extensions to WS-Trust can be translated in a one-to-one fashion into an object that can be parsed TrustBuilder2. TrustBuilder2 can then examine the state of the negotiation, determine the next step, and generate an appropriate response. This response can then be translated into the XML elements described in Section 4.2 and transmitted.

6 Conclusions

Web services are a promising distributed computing paradigm, but fully unlocking their potential requires flexible authorization techniques that can function correctly without a priori knowledge of the users in the system. In this paper, we have shown that the adoption of *trust negotiation* within this realm can occur within the framework provided by existing web services security standards. In particular, we showed that after defining a rudimentary claims dialect—which is fully-compliant with the WS-Trust standard—the WS-Policy and WS-SecurityPolicy standards can be used to define a range of expressive trust negotiation policies. We also showed that WS-Trust's negotiation and challenge framework can be extended to act as a standards-compliant transport mechanism within which trust negotiation sessions can occur.

We also examined the systems aspects of this process and showed that trust negotiation policies specified using the WS-Policy and WS-SecurityPolicy standards can be complied into a format that is suitable for analysis by CLOUSEAU, an efficient policy compliance checker for trust negotiation systems. This not only eases the development of trust negotiation solutions for the web services domain, but shows that it is possible to design a *single* compliance checker—namely CLOUSEAU—that is capable of analyzing Datalog-style policy languages, as well as other industry standard policy languages. Furthermore, we show that the TrustBuilder2 framework

for trust negotiation can be parameterized to act as a trust engine, as described by the WS-Trust standard, that can be used to drive these interactions.

Acknowledgments

This research was supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695 and by Sandia National Laboratories under grant number DOE SNL 541065.

References

[1] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 81–95, May 2005.

[2] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, June 2004.

[3] Elisa Bertino, Elana Ferrari, and Anna Cinzia Squicciarini. $\mathcal{X}$-TNL: An XML-based language for trust negotiations. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 81–84, June 2003.

[4] Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust-$\mathcal{X}$: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, July 2004.

[5] Scott Boag, Don Chamberlain, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon, and (Editors). XQuery 1.0: An XML Query Language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery/`.

[6] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security (CCS)*, pages 134–143, November 2000.

[7] Business process execution language for web services version 1.1. Web page, February 2007. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`.

[8] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. W3C Note, March 2001. `http://www.w3.org/TR/wsdl`.

[9] Amir Herzberg, Yosi Mass, Joris Michaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–14, May 2000.

[10] H. Koshutanski and F. Massacci. Interactive access control for web services. In *Proceedings of the 19th IFIP Information Security Conference (SEC)*, pages 151–166, August 2004.

[11] H. Koshutanski and F. Massacci. An interactive trust management and negotiation scheme. In *Proceedings of the Second International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 139–152, August 2004.

[12] H. Koshutanski and F. Massacci. Interactive credential negotiation for stateful business processes. In *Proceedings of the Third International Conference on Trust Management (iTrust)*, pages 257–273, May 2005.

[13] Adam J. Lee and Marianne Winslett. Towards and efficient and language-agnostic compliance checker for trust negotiation systems. In *Proceedings of the Third ACM Symposium on Information, Computer and Communications Security (ASIACCS 2008)*, March 2008.

[14] Adam J. Lee and Marianne Winslett. Towards standards-compliant trust negotiation for web services. In *Proceedings of the Joint iTrust and PST Conferences on Privacy, Trust Management, and Security (IFIPTM 2008)*, June 2008.

[15] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

[16] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, Hans Granqvist, and (Editors). WS-SecurityPolicy 1.2. OASIS Standard, July 2007. `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/`.

[17] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, Hans Granqvist, and (Editors). WS-Trust 1.3. OASIS Standard, March 2007. `http://docs.oasis-open.org/ws-sx/ws-trust/200512/`.

[18] Anthony Nadalin, Chris Kaler, Ronald Monzillo, Phillip Hallam-Baker, and (Editors). WS-Security Core Specification 1.1. OASIS Standard, February 2006. `http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`.

[19] Jeffrey Schlimmer and (Editor). Web Services Policy 1.2 - Framework (WS-Policy) . W3C Member Submission, April 2006. `http://www.w3.org/Submission/WS-Policy/`.

[20] OASIS UDDI Specifications TC. Web page. `http://www.oasis-open.org/committees/uddi-spec/`,.

[21] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 88–102, January 2000.

[22] Marianne Winslett, Charles Zhang, and Piero Andrea Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, pages 168–179, November 2005.

[23] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies throu gh interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), Feb. 2003.

A Claim Dialect Schema

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://dais.cs.uiuc.edu/claim.xsd"
           xmlns="http://dais.cs.uiuc.edu/claim.xsd"
           elementFormDefault="qualified">

  <xs:element name="Claim">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Attribute" type="xs:string"/>
        <xs:element name="Op">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="LT"/>
              <xs:enumeration value="GT"/>
              <xs:enumeration value="EQ"/>
              <xs:enumeration value="LTEQ"/>
              <xs:enumeration value="GTEQ"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="Value" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Ownership">
    <xs:complexType>
      <xs:attribute name="status" type="xs:boolean" />
    </xs:complexType>
  </xs:element>

</xs:schema>
```

B Negotiation Extension Schema

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://dais.cs.uiuc.edu/negotiation.xsd"
           xmlns="http://dais.cs.uiuc.edu/negotiation.xsd"
           elementFormDefault="qualified">
  <xs:import namespace="http://schemas.xmlsoap.org/ws/2004/09/policy"
             schemaLocation="http://schemas.xmlsoap.org/ws/2004/09/policy/ws-policy.xsd"/>

  <xs:element name="TNInit">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="OwnershipProof" type="xs:String"/>
        <xs:element name="StrategyFamily" type="xs:String" minOccurs="1" maxOccurs="unbounded" />
        <xs:element name="TokenFormat" type="xs:String" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="TrustNegotiationExchange">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="TokenCollection" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="PolicyCollection" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="TokenCollection">
    <xs:complexType>
      <xs:sequence>
        <element ref="Token" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Token">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wst:TokenType"/>
        <xs:element ref="wst:RequestedSecurityToken"/>
        <xs:element name="OwnershipProof" type="xs:String"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="PolicyCollection">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wsp:Policy" minOccurs="1" maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```