

Traust: A Trust Negotiation-Based Authorization Service for Open Systems

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

Jim Basney and Von Welch
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark St.
Urbana, IL 61801
{jbasney, vwelch}@ncsa.uiuc.edu

ABSTRACT

In recent years, trust negotiation (TN) has been proposed as a novel access control solution for use in open system environments in which resources are shared across organizational boundaries. Researchers have shown that TN is indeed a viable solution for these environments by developing a number of policy languages and strategies for TN which have desirable theoretical properties. Further, existing protocols, such as TLS, have been altered to interact with prototype TN systems, thereby illustrating the utility of TN. Unfortunately, modifying existing protocols is often a time-consuming and bureaucratic process which can hinder the adoption of this promising technology.

In this paper, we present Traust, a third-party authorization service that leverages the strengths of existing prototype TN systems. Traust acts as an authorization broker that issues access tokens for resources in an open system after entities use TN to satisfy the appropriate resource access policies. The Traust architecture was designed to allow Traust to be integrated either directly with newer trust-aware applications or indirectly with existing legacy applications; this flexibility paves the way for the incremental adoption of TN technologies without requiring widespread software or protocol upgrades. We discuss the design and implementation of Traust, the communication protocol used by the Traust system, and its performance. We also discuss our experiences using Traust to broker access to legacy resources, our proposal for a Traust-aware version of the GridFTP protocol, and Traust's resilience to attack.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—*access controls, authentication*; K.6.5 [Management of Computing and Information Systems]: Security and Protection; C.2.3 [Computer-Communication Networks]: Network Operations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SACMAT'06, June 7–9, 2006, Lake Tahoe, California USA.
Copyright 2006 ACM 1-59593-354-9/06/0006 ...\$5.00.

General Terms

Security, Management

Keywords

Attribute-based access control, credentials, trust negotiation

1. INTRODUCTION

Due to recent Internet trends—including peer-to-peer networks, grid computing, and corporations restructuring as virtual organizations—large-scale open systems in which resources are shared across organizational boundaries are becoming ever more popular. Making intelligent access control decisions in these systems is a difficult task, as a potentially unbounded number of users and resources exist in an environment with few guarantees regarding pre-existing trust relationships. Traditional access control systems fail to work in these systems either because they cannot scale to such a large user base or make unrealistic assumptions about existing trust relationships in the system. As open systems continue to gain popularity, it is critical that the access control problem be addressed.

Trust negotiation is an active area of research aiming to help solve the problems surrounding authorization in open systems. In trust negotiation, access control decisions are made based on the attributes of the entity requesting access to a particular resource, rather than his or her identity. To determine whether an entity should be granted access to a resource, the entity and resource provider conduct a bilateral and iterative exchange of policies and credentials (used to certify attributes) to incrementally establish trust in one another.

To date, work in trust negotiation has focused primarily on the development of languages and strategies for trust negotiation (from at least eight research groups [1, 3, 5, 9, 10, 14, 16, 23, 26, 30]) and the embedding of trust negotiation into commonly used protocols [11]. These research efforts have shown that the flexible nature of trust negotiation makes it a viable solution to the problem of authorization in open systems. If software engineers could easily redesign all major applications and protocols to support trust negotiation natively, the problem of making authorization decisions in open systems would be solved. Unfortunately, redesigning and restandardizing existing protocols is a time-consuming process. To address this problem, we propose Traust, a stand-alone authorization service that allows for

the adoption of trust negotiation in a modular, incremental, and grassroots manner, providing access to a wide range of resources without requiring widespread software or protocol upgrades.

In our approach, a collection of Traust servers act as brokers for the security tokens needed to gain access to the resources located in a given security domain. The format of these tokens is not restricted by Traust; tokens can be of any format, including (username, password) pairs, Kerberos tickets, SAML assertions, and X.509 certificates. Clients contact Traust servers and negotiate for access tokens for logical or physical resources including network servers, RPC methods, and organization-wide roles. The Traust service also provides clients in the system with an opportunity to establish trust in the service prior to the disclosure of their (potentially sensitive) resource access requests.

The Traust system was designed explicitly to meet the needs of large-scale open systems. In particular, the Traust system:

- uses current prototype trust negotiation systems (such as Trust-X [3] or TrustBuilder [27]) to allow clients to establish bilateral trust with previously-unknown resource providers on-the-fly and negotiate for access to new system resources at runtime;
- integrates transparently with newer, trust-aware resources while still maintaining compatibility with and allowing increased access to legacy resources;
- can broker access tokens in any format for any size security domain, ranging from single hosts (e.g., in peer-to-peer systems) to entire organizations;
- has policy maintenance overheads that scale independently of the number of users in the system and the rates at which users join and leave the system.

The rest of this paper is organized as follows. In Section 2, we provide an overview of trust negotiation and discuss related work in this area. Section 3 highlights the defining characteristics of open systems; these are then used to derive several important requirements for authorization systems designed for these environments. Sections 4 and 5 present the details of the Traust system architecture and resource access protocol, respectively. In Section 6 we describe our implementation of the Traust system and its performance, discuss our experiences using Traust to broker access to existing legacy services, and comment on our proposal for a Traust-aware GridFTP client and server. In Section 7 we discuss how Traust meets the requirements identified in Section 3 and present a security analysis of the Traust system. We conclude and discuss potential directions for future work in Section 8.

2. RELATED WORK

In this section, we first present an overview of trust negotiation and its application to open systems. We then discuss current research efforts in this area and their relationship to Traust.

2.1 Trust Negotiation

Trust negotiation is a technique that has been proposed to address the scalability limitations of existing access control solutions when used in the context of open systems. In

trust negotiation, the access policy for a resource is written as a declarative specification of the attributes that an authorized entity must possess in order to gain access to the resource. In these systems, credentials and policies are also considered resources, so sensitive credentials and policies can be protected by release policies of their own. In this way, an access request leads to a bilateral and iterative disclosure of credentials and policies between the user and resource provider. During this process, trust is established incrementally as more and more sensitive credentials are exchanged.

As an example trust negotiation, consider the case in which a user, Alice, wishes to access a service provided by Bob. After Alice requests access to Bob's service, Bob discloses the access policy for his service, which states that in order to use the service, Alice must disclose her digital student ID. To protect herself from identity theft, Alice is only willing to disclose her student ID credential to members of the Better Business Bureau (BBB), so rather than disclose her student ID credential, Alice sends Bob this release policy. Bob is in fact a member of the BBB and is willing to disclose this credential to anyone. This satisfies Alice, who discloses her digital student ID credential to Bob and is then granted access to Bob's service.

Access control systems based on trust negotiation are natural candidates for use in open systems. Allowing resource access policies to be specified based on the attributes of authorized users circumvents the scalability problems associated with maintaining identity- or organization-based ACLs as the size of the system increases. In addition, trust negotiation allows mutually distrustful parties to gain trust in one another incrementally and bilaterally in a privacy-preserving manner.

2.2 Current Research

In recent years, trust negotiation has been an active area of research within the security community. Recent research in trust negotiation has focused on a number of important issues including languages for expressing resource access policies (e.g., [1, 2, 9, 16]), protocols and strategies for conducting trust negotiations (e.g., [3, 14, 15, 30]), and logics for reasoning about the outcomes of these negotiations (e.g., [5, 28]). The foundational results presented in these works have also been shown to be viable access control solutions for real-world systems through a series of implementations (such as those presented in [3, 13, 27]) which demonstrate the utility and practicality of these theoretical advances.

Implementations of trust negotiation typically provide a means of parsing policies, handling certified attributes, and determining policy satisfaction. Existing trust negotiation implementations have been successfully embedded in several commonly used applications and protocols (for a number of examples, see [11, 12]). Unfortunately, trust negotiation is in many ways fundamentally different from existing access control solutions and integrating these implementations with existing protocols has been a challenging process involving the modification of standardized protocols. A detailed discussion of modifications made to TLS to support trust negotiation-based access control for the World Wide Web is presented in [11]. While this clearly demonstrates the utility of trust negotiation, revising the protocols needed to access every resource used in open computing systems would be a daunting task.

Traust was designed to provide an easier migration path for the adoption of trust negotiation. Traust servers act as reference monitors that use existing trust negotiation implementations to determine which users are authorized to access resources within their protection domains. Authorized users are issued access tokens by the Traust server which are encoded in formats understood by existing applications. Traust servers are authorization brokers that effectively use trust negotiation to control access to legacy resources without requiring protocol or software upgrades at these endpoints. Traust can also be integrated directly with newer trust-aware applications, thereby making it a viable long-term access control solution for open systems rather than simply a short-term fix. In the remainder of this paper, we discuss the design and implementation of the Traust system.

3. DESIGN REQUIREMENTS

In designing Traust, our goal was not only to provide a migration path for the integration of trust negotiation technologies into existing open systems, but also to provide a general-purpose authorization service which meets the needs of open systems to the highest degree possible. To this end, we now explore the defining characteristics of open systems. We then use these characteristics to derive a set of functional requirements which should be satisfied by any access control solution designed for use in open systems.

In large-scale open systems, resource providers often wish to realize the competitive advantages offered by allowing qualified outsiders access to some of their resources under certain conditions. Due to the large number of potential users in these environments (e.g., all users with a valid student ID), we cannot assume that resource providers will know a priori the identities of all authorized clients that might possibly wish to access their resources. In addition, we cannot assume that clients will know the set of resource providers that they might wish to interact with prior to the start of these interactions. Given that there are compelling business reasons for resource providers to permit all possible qualified users to access their resources, we can immediately recognize four important requirements that must be satisfied by any authorization system designed for use in open computing systems.

Bilateral trust establishment To enable effective resource sharing, we cannot require pre-existing trust relationships between clients and resource providers; it is important to allow these entities to establish trust relationships with one another at runtime.

Runtime access policy discovery In large-scale open systems, clients cannot be expected to know a priori the access policies protecting resources of interest. Authorization systems used in these environments should allow clients to discover these policies as they become relevant.

Privacy preservation To protect clients and resource providers from malicious entities, their interactions should reveal as little information as possible. Entities should have some ability to control their disclosure of sensitive information, including their objectives, policies, identities, and attribute information.

Scalability Authorization systems used in open computing systems should be scalable both in terms of maintenance overhead and size of protection domain. Access policies should scale well in spite of a potentially unbounded number of users joining and leaving the system, while still maintaining an appropriate level of expressiveness. To accommodate the heterogeneity of these systems, the service should be light-weight enough for a single user (e.g., a peer-to-peer client) to deploy on her local machine, yet robust enough to meet the demands of a large security domain.

In addition to these four requirements, it is important to include another, more practical, property:

Application Support Incorporating a new authorization service into existing open systems should not require a complete redesign of deployed applications, protocols, or the network infrastructure. The authorization system should support tight interaction with newer applications designed to leverage its features explicitly, but also remain accessible to clients who wish to access legacy applications.

We do not make the claim that the above list of requirements is complete, as completeness will always depend on the *specific* needs of a system's participants. However, a system embodying these five requirements will allow resource providers to ensure that their resources (e.g., data, computational clusters, or other services) are available to as many *qualified* users as possible without compromising the security of the resource itself. Such a system will also enable users to maximize their productivity by discovering resources at runtime and dynamically gaining access to them. In Section 7.1, we revisit these requirements and discuss how Traust satisfies each of them.

4. TRAUST SYSTEM ARCHITECTURE

Figure 1 illustrates the Traust system architecture. In the remainder of this section, we describe each component in greater detail.

Traust Servers In our system, Traust servers act as brokers for the access rights to a set of resources in their security domain. A Traust server contains a protocol interpreter that is responsible for carrying out the steps of the Traust resource access protocol (discussed in Section 5.3) and has some means of interacting with its Trust Negotiation Agent (or Agents). Each Traust server also maintains a repository of access tokens used to grant access to the resources that it protects; these tokens are issued to authorized users who negotiate for access to the resources protected by the Traust server.

Traust Clients A Traust client is a process designed to acquire access tokens for resources of interest to its owner. Like a Traust server, a Traust client also contains a protocol interpreter and a means of contacting its Trust Negotiation Agent (or Agents). For systems in which resource requests could themselves be considered sensitive (e.g., requests to access classified data), Traust clients may have a local resource classifier which can be used to determine the sensitivity classification (or classifications) of a particular resource request and

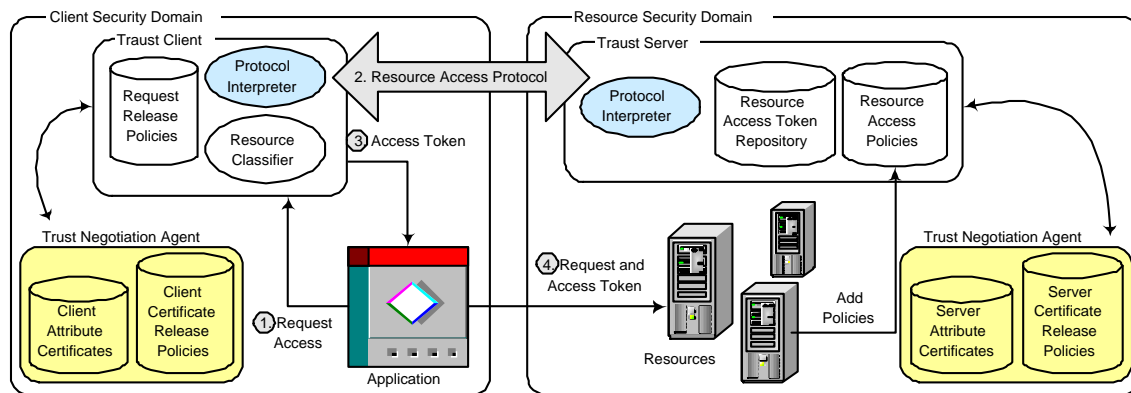


Figure 1: Traust system architecture

identify its corresponding release policy (or policies). For maximum flexibility, a Traust client can be accessed directly by a user or by a Traust-aware application. Further information regarding these two modes of operation is presented in Section 6.

Trust Negotiation Agents Traust clients and servers require access to one or more Trust Negotiation Agents. A Trust Negotiation Agent is responsible for understanding the protocol used for trust negotiation (e.g., the Trust-X [3], TTG [25], or TrustBuilder [27] protocol) and carrying out trust negotiation sessions on behalf of the client or server processes that own it. In addition, a Trust Negotiation Agent manages its owners' attribute certificates and their corresponding release policies.

Logically, a Trust Negotiation Agent is part of the Traust client and server applications, though it need not run on the same physical machine and can be a shared resource for all of a user's processes. This allows for increased flexibility, as the overheads of running the agent can be shared across multiple processes. Allowing a Traust server to access multiple Trust Negotiation Agents also permits load-balancing during periods of high traffic.

Access Token Repository Each Traust server maintains a repository of access tokens that can be used to access the services that it protects. This repository is not a repository in the traditional sense, which implies that it contains a static collection of tokens. Rather, the repository may contain static tokens, but may also contain instructions on obtaining or creating new tokens at runtime. For instance, the repository may create new local accounts used to access resources that it protects, acquire Kerberos tickets, generate SAML assertions, or be delegated proxy certificates from a MyProxy [17] server.

Resources Resources are the logical and physical objects that Traust servers broker the access rights to. Some examples of resources include networks, individual machines, services (e.g., web sites or file servers), RPC methods, web services, or organization-wide role memberships.

5. PROTOCOL OVERVIEW

In this section, we present an overview of the communication protocol used in the Traust system and discuss the ways in which Traust components interact during the execution of this protocol. We focus our attention on message semantics and permissible sequences of messages; the full details of the Traust protocol, including message contents and formats, can be found in the Traust protocol specification.¹

5.1 Session Security

All communications between a client and Traust server occur inside of a TLS [7] tunnel used to provide confidentiality and integrity for the session. The tunnel itself is not used to provide any notion of authentication or authorization; one or more trust negotiation sessions are used for this purpose. We discuss these trust negotiation sessions in greater detail in Section 5.3.

5.2 Message Types

Messages in the Traust protocol can be divided into two categories: functional messages and trust establishment messages. The functional messages and their replies allow a Traust client to make requests of a Traust server and be provided with information in return. The current version of the Traust protocol supports two types of functional messages: *Get Information* and *Resource Request*.

Get Information (GI) The GI message allows Traust clients to request public meta-data regarding a Traust server with which they have an established connection. The server's response to this message may include information such as software and protocol versions, a "message of the day," administrative points of contact, or other site-specific information. A GI request may only be sent by the client at the start of a Traust session.

Resource Request (RR) The RR message and its corresponding response embody the main functionality of the Traust service. RR messages contain a URI and a series of optional (attribute, value) pairs describing a resource that the Traust client wishes to acquire access tokens for. This naming system is flexible enough to

¹The entire Traust protocol specification is available at <http://dais.cs.uiuc.edu/traust/standards.html>.

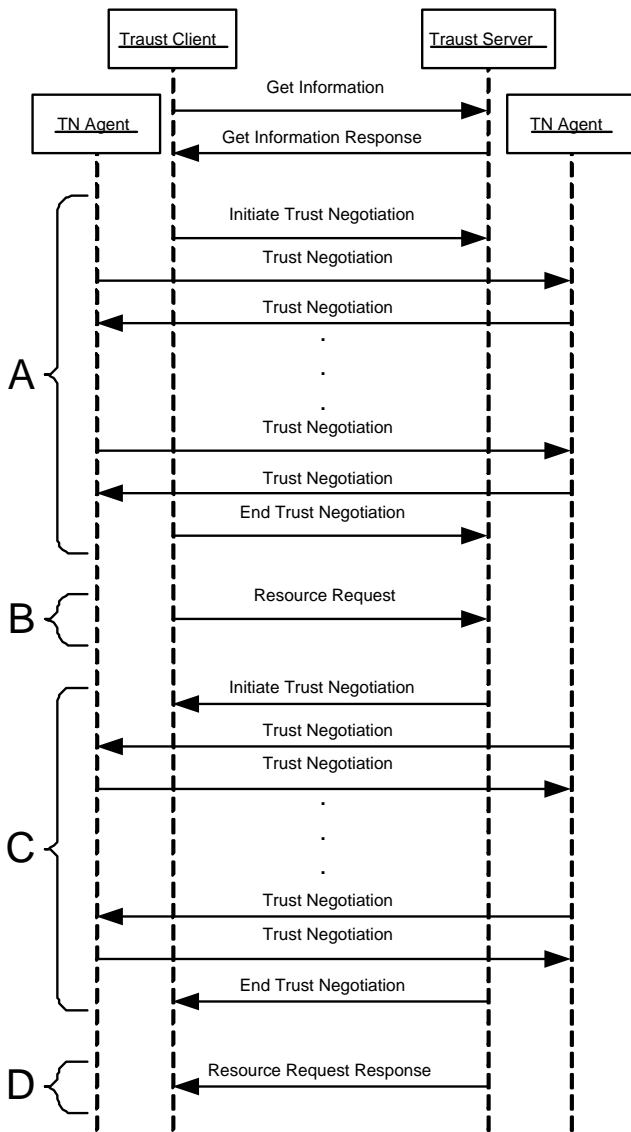


Figure 2: The Traust resource access protocol

specify a wide variety of resources, including entire networks, enterprise-wide roles, or individual hosts, services, or method calls. The server response to this message contains either a failure notification or a collection of access tokens that can be used to access the requested resource.

To control the flow of sensitive information between clients and servers in the system, Traust supports three trust establishment message types: *Initiate Trust Negotiation*, *Trust Negotiation*, and *End Trust Negotiation*.

Initiate Trust Negotiation (ITN) This message serves as a flag to indicate that a new trust negotiation session is about to begin. After receiving an ITN message, the Traust client (or server) will forward subsequent messages to one of its associated Trust Negotiation Agent processes for processing until an *End Trust Negotiation* message is received.

Trust Negotiation (TN) TN messages are used to encapsulate a trust negotiation session carried out between the Trust Negotiation Agent processes of the Traust client and Traust server. The policies and credentials exchanged between parties in the Traust system are encoded in the body of these messages. Several rounds of TN messages may be required for the initiating party to determine whether an acceptable level of trust has been gained in the responding party.

End Trust Negotiation (ETN) The ETN message serves as a flag to indicate that a trust negotiation session has just completed. Upon receiving an ETN message, the receiver will cease forwarding subsequent messages to their Trust Negotiation Agent.

Next, we describe how these messages are ordered to form the communication protocol used in the Traust system.

5.3 Resource Access Protocol

At a high level, the Traust resource access protocol maps users' attributes into access tokens that are meaningful in the local security domain of the resource that is to be accessed. This protocol takes place in five stages: local classification, server trust establishment, request disclosure, client trust establishment, and response.

Prior to establishing a connection to a Traust server, the user's Traust client is provided with the description of a resource that the user wishes to negotiate for access to. This description may be generated explicitly by the human user (e.g., after reading a web page describing how to access a legacy service protected by a Traust server) or generated on-the-fly by a client application interacting with a Traust-aware resource. During the *local classification* stage, this resource description is examined using a local content classifier to determine its sensitivity classification or classifications. The Traust client then maps these sensitivity classifications into release policies that will be used in the server trust establishment phase.

During *server trust establishment*, indicated by the label 'A' in Figure 2, the Traust client initiates zero or more content-triggered trust negotiation sessions [10] with the Traust server—one for each release policy discovered during local classification. Alternatively, the client could initiate a single negotiation using a new policy derived from some function of these release policies. This process determines whether the Traust server is trustworthy enough to receive the resource request issued to the Traust client and prevents inadvertent disclosure of sensitive requests to unauthorized Traust servers.

Each trust negotiation session in the server trust establishment phase is initiated by the client sending an *Initiate Trust Negotiation* message to the server. The client's Trust Negotiation Agent then conducts an iterative exchange of *Trust Negotiation* messages with the server's Trust Negotiation Agent, reporting the results of this negotiation back to the Traust client. The Traust client terminates this phase by sending an *End Trust Negotiation* message to the Traust server. Should the client fail to establish trust in the server during this phase of the protocol, the Traust client closes its connection with the server and reports a failure to the user.

If the Traust client establishes trust in the server, the Traust session enters the *resource disclosure* stage. At this point, the Traust client sends a *Resource Request* message

$$(GI_CGI_S)?(ITN_C(TN)^*ETN_C)^*RR_C(ITN_S(TN)^*ETN_S)?RR_S$$

Figure 3: A regular expression describing successful executions of the resource access protocol

to the Traust server describing the resource that the user wishes to access. This disclosure is indicated by the label ‘B’ in Figure 2.

Upon receiving the Traust client’s Resource Request message, the Traust server examines it to determine the access policy that protects the requested resource. The server then begins the *client trust establishment* phase, indicated by the label ‘C’ in Figure 2, by sending an Initiate Trust Negotiation message to the client. The Traust server’s Trust Negotiation Agent then carries out a negotiation with the client’s Trust Negotiation Agent via an iterative exchange of Trust Negotiation messages. When the negotiation is over, the Traust server sends an End Trust Negotiation message to the Traust client to indicate this fact.

In the *response* phase, indicated by the label ‘D’ in Figure 2, the Traust server indicates the status of the resource access protocol. If the server failed to establish trust in the client, the client is sent a failure notification in the Resource Request response message. If the Traust server did establish trust in the client, however, it obtains the access tokens needed for the client to access the requested resource and passes these tokens to the client in the Resource Request response message. Obtaining an access token could be as simple as looking up a static token, or could involve generating a new local account or interacting with an external service (e.g., Kerberos, MyProxy, or CAS [19]) to obtain the needed access tokens.

To help detect certain types of attacks, it is important to differentiate between valid and invalid executions of the resource access protocol. Figure 3 is a regular expression describing successful executions of the Traust resource access protocol. The message abbreviations are those used in Section 5.2 and the subscripts indicate whether a particular message was sent by the client (*C*) or server (*S*). Note that we include the optional transmission of a Get Information message and its response, though it was not discussed in this section. Any sequence of messages not described by this expression is considered invalid; a correctly functioning participant in the resource access protocol should immediately close any connection upon which an invalid execution has been detected.

6. IMPLEMENTATION

In this section, we discuss our implementation of the Traust system. In addition, we discuss our experiences using Traust to broker access to legacy resources (e.g., password-protected web sites), comment on our proposal for a Traust-aware GridFTP client and server, and address the performance of our prototype implementation.

6.1 Implementation Details

We have developed a prototype implementation of the Traust service using the Java programming language. We provide a client API that can be embedded into applications that wish to interact directly with a Traust server. In addition, we have developed both command line and graphical Traust clients which allow human users to interact with

a Traust server to request access tokens for legacy services whose clients do not natively support Traust. We also provide an extensible resource classification API which allows users to develop custom request sensitivity classifiers. Because defining these types of classifiers can be difficult, a user’s organization (e.g., their employer or credential issuer) is likely to supply them with the classifiers for sensitive requests. In our implementation, a resource classifier based on substring matching is used by default.

Our server implementation provides an extensible API which can be used to interface with a wide variety of access token repositories. We have developed a simple, yet flexible, token repository that allows the server to obtain the tokens needed to access a given resource by either (1) accessing tokens stored directly in the repository, (2) referencing files located on the local file system, or (3) interfacing with external processes. We have used the latter mechanism to generate one-time-use passwords, delegate X.509 proxy certificates, and create temporary local accounts.

Both the client and the server currently utilize Trust Negotiation Agents based on the TrustBuilder framework for trust negotiation [27]. TrustBuilder currently supports the use of X.509 attribute certificates for credentials and the IBM Trust Policy Language [9] for access policy specification, with future support for other policy languages and credential types. TrustBuilder has been successfully integrated with a number of protocols and applications [10, 11, 20], making it a good choice for use in the Traust system. In the future, we plan to extend the Traust resource access protocol to allow for the use of trust negotiation agents other than TrustBuilder.

6.2 Usage with Legacy Resources

In some computing environments, it is considered acceptable to require that clients manually acquire resource access tokens prior to using networked services. For instance, at many universities, users wishing to access student records must first acquire a Kerberos ticket using a stand-alone client application (e.g., *kinit*). For these environments, we have developed command line and graphical Traust clients that allow users to manually request access tokens for legacy services that do not natively support Traust interaction. Figure 4 shows a screen shot of our graphical Traust client.

As an example of how Traust might be used in this type of environment, consider the case of a rescue dog handler who hears a newscast about a building that has collapsed and wishes to help in the recovery effort. The newscast gives the URL of a web-based information portal that will be used to coordinate the recovery effort. The user browses to this web site and is presented with a login form and resource descriptor to pass into his Traust client that will allow him to negotiate for a temporary login and password to the portal. The Traust interaction allows the client to establish trust in the server (e.g., that the server is a state-sponsored disaster response coordinator, not a hoax) and allows the server to verify the user’s credentials (e.g., that he is a certified rescue dog handler with up-to-date vaccinations, not a news reporter looking for a hot story). The Traust server then



Figure 4: The graphical Traust client

returns a temporary login and password for the web site, which the client application displays to the user.

We have built a testbed information portal for this application and used Traust to allow previously unknown, but qualified, users to obtain authorization to access the information contained within. In addition, Traust has been used in a similar fashion to issue X.509 proxy certificates that control access to a file server.

6.3 Traust-Aware Resources

In addition to using Traust to control access to legacy resources, we wish to allow for the development of services that support Traust natively. These applications can embed Traust interactions in their access protocols, allowing users' client applications to carry out any necessary Traust interactions without requiring the user to initiate this process. As an example of how to permit this form of tight interaction with Traust, we have proposed two modifications to GridFTP, a secure mass data transfer protocol used heavily in the context of grid computing.²

The first of our proposed modifications would allow a user's GridFTP client application to query the GridFTP server (prior to login) for the name of the Traust server brokering access to this server. The client application could then execute the Traust resource access protocol with this server without user intervention. If the Traust server authorizes the client to access the GridFTP server, the Traust server would issue the client an X.509 proxy certificate that the client could use to log into the the GridFTP server using the existing Grid Security Infrastructure [24].

Our second modification defines a simple syntax for access "hints" that could be provided by a GridFTP server. In the event that a client command fails due to insufficient access rights, the server could embed an access hint in the error message returned to the client application. These hints identify a Traust server to contact, and a corresponding resource request. A Traust-aware GridFTP client could then, without user intervention, execute the resource access protocol with this Traust server in order to obtain a new X.509 proxy certificate. This certificate could be used to re-authenticate to the GridFTP server and successfully retry the previous request without terminating the current session. This would allow GridFTP servers to enforce the prin-

²The complete details of our proposed modifications are available at <http://dais.cs.uiuc.edu/traust/standards.html>.

ciple of least privilege [21], as clients' access rights could be changed as they perform different operations on the server.

6.4 Performance

We now comment on the performance of our implementation in two representative usage scenarios. All averages reported in this section were calculated over 10 trials executed between a 2.8GHz Pentium 4 with 1GB RAM running Windows XP SP2 and a 2.5GHz Pentium 4 with 512MB RAM running Linux. In the first scenario, the client releases its resource request to the Traust server without requiring a trust negotiation. The Traust server then initiates a single-round trust negotiation with the client in which the client demonstrates proof of ownership of one attribute certificate. This case is indicative of Traust interactions that might be seen in corporate environments where users are asked to show their digital employee ID card or role certificate to gain access to a particular resource. We ran this scenario across our department's network at midday and found that, on average, it executed in 2.77 seconds with a standard deviation of 0.18 seconds. The two major components of this time are connection establishment (1.12 seconds) and creating the client Trust Negotiation Agent and carrying out the trust negotiation (1.33 seconds).

The second scenario uses the disaster response information portal discussed in Section 6.2. In this scenario, the client is only willing to disclose his access request to Traust servers that can prove that they are operated by a state-sponsored disaster response coordinator, and uses the server trust establishment phase of the resource access protocol to enforce this. The Traust server is able to prove ownership of an attribute credential indicating this fact, which satisfies the client, who then discloses his request for access to the information portal.

At this point, the server requests that the client prove that he is a certified rescue dog handler, is over the age of 18 (by showing a state-issued driver's license), has a recent tetanus vaccination (the record of which is issued by a state-certified board of health), and that his dog has a recent rabies vaccination (the record of which is issued by a state-certified county). In response to this request, the client demonstrates proof of ownership of his rescue dog handler certificate and discloses the release policies protecting his driver's license and both vaccination records. The release policy protecting his driver's license requires that the server disclose a privacy policy issued by an accrediting organization, while the release policy protecting both vaccination records requires that the server prove that it is operated by a department of some U.S. state. The Traust server is willing to disclose this information, at which point the client sends over the remaining credentials required by the server. In all, these two trust negotiations took place over three rounds and involved the disclosure of nine credentials (including supporting credentials for certification chains). On average, this scenario executed in 4.04 seconds over our department's network at midday with a standard deviation of 0.12 seconds. The main components of this time are connection establishment (1.12 seconds), creating the client Trust Negotiation Agent and carrying out the first negotiation (1.58 seconds), and the second trust negotiation (1.34 seconds).

Clearly, executing the Traust resource access protocol takes longer than using a more traditional means of acquiring resource access tokens (e.g., obtaining a Kerberos ticket).

However, this comparison means very little, as traditional access control systems cannot be used in open systems environments since the identities of authorized users may not be known a priori, requiring users to resort to out-of-band methods to gain access (e.g., sending written requests to resource providers). Additionally, the client used in these tests created a new Trust Negotiation Agent at each invocation, a process which takes 0.93 seconds on average; configuring the client to use a stand-alone Trust Negotiation Agent would eliminate this overhead. Further, to the best of our knowledge, the TrustBuilder system has not undergone a performance evaluation study, so there exists opportunity for optimization within that system. The benefits of allowing previously unknown users to negotiate for access to resources outweigh the modest cost of the negotiation.

Currently, studying the scalability of Traust as the number of concurrent connections increases would have little value. The policy engines used by prototype trust negotiation implementations such as TrustBuilder are highly unoptimized and would skew any measured results. However, we plan to conduct such a study once high-performance policy evaluators such as CPOL [6] are integrated with existing trust negotiation systems.

7. DISCUSSION

In this section, we discuss the security properties of the Traust system. First, we describe the ways in which Traust meets the needs of large-scale open systems by addressing each of the requirements presented in Section 3. We then present an informal security analysis of Traust and address possible attacks against the system.

7.1 Requirements Revisited

Section 3 introduced five requirements for authorization systems to be used in open systems: bilateral trust establishment, runtime access policy discovery, preservation of privacy, scalability, and application support. The Traust system architecture and resource access protocol were designed to address these goals from the start. Bilateral trust establishment and runtime access policy discovery are attained through the use of trust negotiation in the server and client trust establishment stages of the resource access protocol. To help preserve privacy, these negotiations can leverage negotiation strategies that limit the disclosure of sensitive credentials. In environments where some requests themselves could be considered private, clients may also enforce their own request release policies. Traust's use of trust negotiation implies that access policies are specified in terms of the attributes that an authorized user must possess, thus policy maintenance overheads scale independently of the number of users joining and leaving the system. The performance of the Traust service prototype is reasonable (roughly 4 seconds on average for a complex interaction). Lastly, Traust integrates with both legacy and Traust-aware resources.

7.2 Security Evaluation

Though Traust adequately addresses the five functional requirements discussed in Section 3, these properties say very little about the security of the system. In this section, we discuss the security properties of the Traust system and address several potential attacks against Traust.

7.2.1 Session Security

In the Traust system, session security is provided through the use of the TLS protocol. In [7], the authors present a security analysis of the TLS protocol under the assumption of an active attacker [8] with the ability to intercept, modify, delete, and replay messages sent over the communication channel. In the case that the public key of one party in the protocol is authenticated, the authors show that the TLS channel is secure against man-in-the-middle attacks, thus the two parties can be assured of the confidentiality and integrity of the messages transmitted using TLS. In situations where a Traust server is run by an organization such as a university, research center, or corporation, the server can be issued a certified public key much in the same way that World Wide Web servers are issued certified keys today. In these cases, the security evaluation presented in [7] applies to the session security of Traust.

In environments where neither the client nor the server has a certified public key (e.g., peer-to-peer networks), the TLS protocol is vulnerable to a man-in-the-middle attack during session establishment. The implication of this attack is that an unauthorized third party can read and alter messages sent through the TLS tunnel, unknown to the client and server. The SSH [29] protocol is subject to the same such attack, as it is rarely the case that SSH servers have certified public keys. In SSH, the threat of this attack is usually mitigated by caching previously-used public keys. In this way, unless the man-in-the-middle attack occurs during the first connection between the client and server, it can be detected, as the cached public key of the legitimate server and the public key returned by the man-in-the-middle will not match. We argue that the threat of this attack can be reasonably mitigated in Traust by using the same practices as are used in SSH. As in SSH, however, highly sensitive non-certified public keys should be verified out-of-band to prevent this attack. Given that it is possible to prevent man-in-the-middle attacks against the TLS protocol, we argue that the security analysis presented in [7] ensures that messages exchanged during the Traust resource access protocol can be viewed only by their intended recipients.

7.2.2 Single Point of Attack

In addition to attacks on the Traust resource access protocol, we must also consider attacks on the Traust server itself. If a single Traust server brokers access tokens for a large number of resources, it will be an appealing target for attack, as a successful attacker could possibly gain access to a large number of resources by compromising a single Traust server. We now discuss several potential solutions to this problem.

Small protection domains. The Traust server that controls access to a given resource can be run on the same physical machine as the resource itself. In this case, a compromise of the machine that the Traust server is running on only grants the attacker the access tokens needed to access the single resource that the server was protecting. In addition, these tokens are of no value, as the attacker implicitly gains access to that resource by compromising the machine on which the resource is located.

This Traust server configuration model clearly prevents an attacker from gaining access to multiple resources by compromising a single node, and motivates the use of Traust in peer-to-peer systems. We next consider two arrangements of

Traust servers for use in organizations wishing to maintain a more structured organizational model.

Hierarchical arrangement. Here we consider arranging the Traust servers protecting access to an organization’s resources in a hierarchical fashion. In this model, the Traust servers at the upper level of the hierarchy broker access rights for a large number of low-sensitivity resources. As we proceed down the hierarchy, Traust servers broker access to fewer, but more sensitive, resources.

In addition to the distribution of access rights discussed above, high-level Traust servers also know which lower-level servers broker access rights to other resources in the network. This knowledge allows higher-level servers to redirect client traffic to an appropriate lower-level server, making the organizational infrastructure easier to navigate for the client. We believe that this allows an adequate trade-off between the granularity at which Traust servers are deployed and the consequences of compromising one of these servers.

Secret sharing. For organizations not willing to pay the administrative costs associated with maintaining a hierarchy of Traust servers, we now discuss a protection strategy based on secret sharing [4, 22]. In this model, an organization deploys multiple Traust servers, the exact number of which is determined by the organization’s unique needs. A client wishing to access a given resource contacts one of these servers and carries out the resource access protocol to attempt to gain access. Rather than being returned the token needed to access that resource, an authorized client is given a *share* of the access token and a list of other Traust servers. Upon completing the resource access protocol with a preset threshold, k , out of the n Traust servers, the client will have the ability to reconstruct the access token. In this model, an adversary needs to compromise multiple Traust servers to gain access to any resource. Further research is required to investigate how to securely manage the attribute certificates replicated across multiple Traust servers.

In both the hierarchical and secret sharing Traust server deployment models, Traust server administrators must keep several things in mind. To reduce the number of potential vulnerabilities that could be used to compromise a Traust server, only a minimal set of services should be configured. For instance, in [17] the authors suggest that a MyProxy server be run on a “tightly secured host (e.g., comparable to a Kerberos Domain Controller)”; this minimum precaution must also be taken to protect any Traust server brokering access to highly valuable resources. Additionally, if multiple Traust servers are to be deployed, their hardware and software configurations should be as heterogeneous as possible [31, 18] to prevent the threat of a single exploit compromising multiple Traust servers.

Note that it is possible to compose the hierarchical and secret sharing Traust server deployment models to meet the needs of a particular organization or resource provider. This flexibility allows administrators to effectively manage the trade-offs that exist between server maintenance overheads, the arrangement of servers within an organization, and the effects of compromising a Traust server.

7.2.3 Denial of Service

The potentially centralized nature of a Traust deployment along with the relatively heavyweight process of trust nego-

tiation make Traust servers interesting targets for denial of service attacks. To prevent malicious users from extending the number of rounds required to reach a decision during a trust negotiation, Ryutov et. al integrate TrustBuilder with the GAA-API and leverage the GAA-API’s mechanisms for responding to changes in system context [20]. They show how negotiation strategies and policies can be adapted at runtime in response to suspicion of denial of service, thereby increasing system availability. Production implementations of the Trust Negotiation Agent used by the Traust server should use a mechanism such as this to help mitigate the threat of denial of service attacks based on trust negotiation.

Another, more standard, technique to help reduce the risk of denial of service attacks on Traust servers is replication. Since a Traust server can be decoupled from the resources to which it brokers access, high-traffic Traust servers can be replicated to prevent them from becoming bottlenecks. An organization that wishes to both allow their Traust servers to remain available during denial-of-service attacks and prevent the compromise of some number of Traust servers from allowing unauthorized access to their resources could use a k -of- n secret sharing scheme as described above. This scheme ensures that as long as k Traust servers are available, authorized users can obtain the tokens necessary to access resources provided by the organization.

8. CONCLUSIONS & FUTURE WORK

In this paper, we discussed the design and implementation of the Traust authorization service. Traust was designed to enable trust negotiation, a promising new authorization technology designed for open systems, to be integrated with existing protocols and applications used in open systems without requiring the restandardization or upgrade of existing protocols. In addition to providing a migration path for the adoption of trust negotiation, the Traust service was designed to be a long-term access control solution for open systems. We described the Traust architecture and resource access protocol in detail, discussed our implementation and experiences using the Traust service, and presented a security evaluation of Traust.

In the future, we plan to examine how Traust can be extended to support third-party negotiations for the purposes of obtaining new attribute certificates at runtime. We can imagine many situations in which a Trust Negotiation Agent is asked to show an attribute certificate that it does not possess, but could likely obtain. Extending Traust to support attribute certification hints would allow one Trust Negotiation Agent to tell another how to use Traust to locate attribute certificates that it does not currently possess. This however, would also allow malicious entities to waste the time of their negotiation partners, making this an interesting area to investigate. In addition, we plan to further explore how to securely manage the access tokens stored in the repository of a Traust server and extend the Traust resource access protocol to allow parties to use trust negotiation agents other than TrustBuilder.

Acknowledgments

This research was supported by NCSA and by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695. Lee was also supported in part by a Motorola Center for Communications graduate fellowship.

9. REFERENCES

- [1] M. Y. Becker and P. Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.
- [2] E. Bertino, E. Ferrari, and A. C. Squicciarini. X-TNL: An XML-based language for trust negotiations. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '03)*, 2003.
- [3] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-X: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, Jul. 2004.
- [4] G. R. Blakley. Safeguarding cryptographic keys. In *AFIPS Conference Proceedings*, volume 48, pages 313–317, 1979.
- [5] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *7th ACM Conference on Computer and Communications Security*, pages 134–143, 2000.
- [6] K. Borders, X. Zhao, and A. Prakash. CPOL: High-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, Nov. 2005.
- [7] T. Dierks and C. Allen. The TLS protocol version 1.0. IETF Request for Comments RFC-2246, Jan. 1999.
- [8] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, Mar. 1983.
- [9] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, May 2000.
- [10] A. Hess, J. Holt, J. Jacobson, and K. E. Seamons. Content-triggered trust negotiation. *ACM Transactions on Information System Security*, 7(3), Aug. 2004.
- [11] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith. Advanced client/server authentication in TLS. In *Network and Distributed Systems Security Symposium*, Feb. 2002.
- [12] Internet security research lab—projects. Web Page, May 2005. (<http://isrl.cs.byu.edu/TrustBuilder.html>).
- [13] H. Koshutanski and F. Massacci. Interactive access control for web services. In *19th IFIP Information Security Conference (SEC)*, pages 151–166, Aug. 2004.
- [14] H. Koshutanski and F. Massacci. Interactive trust management and negotiation scheme. In *2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 139–152, Aug. 2004.
- [15] H. Koshutanski and F. Massacci. Interactive credential negotiation for stateful business processes. In *3rd International Conference on Trust Management (iTrust)*, pages 257–273, May 2005.
- [16] N. Li and J. Mitchell. RT: A role-based trust-management framework. In *Third DARPA Information Survivability Conference and Exposition*, Apr. 2003.
- [17] J. Novotny, S. Tuecke, and V. Welch. An online credential repository for the grid: MyProxy. In *Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, Aug. 2001.
- [18] A. J. O'Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *11th ACM Conference on Computer and Communications Security*, Oct. 2004.
- [19] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and C. Tuecke. A community authorization service for group collaboration. In *IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.
- [20] T. Rytov, L. Zhou, C. Neuman, T. Leithead, and K. E. Seamons. Adaptive trust negotiation and access control. In *10th ACM Symposium on Access Control Models and Technologies*, Jun. 2005.
- [21] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.
- [22] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.
- [23] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *2nd ACM Workshop on Formal Methods in Security Engineering (FMSE 2004)*, pages 45–55, Oct. 2004.
- [24] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, Jun. 2003.
- [25] W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *Third IEEE International Workshop on Policies for Distributed Systems and Networks*, Jun. 2002.
- [26] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, Jan. 2000.
- [27] 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*, 6(6):30–37, Nov./Dec. 2002.
- [28] M. Winslett, C. Zhang, and P. A. Bonatti. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, Nov. 2005.
- [29] T. Ylonen and C. Lonvick. SSH transport layer protocol. IETF Network Working Group Internet-Draft, Mar. 2005. (<http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-24.txt>).
- [30] 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 and System Security*, 6(1), Feb. 2003.
- [31] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao. Heterogeneous networking: A new survivability paradigm. In *2001 Workshop on New Security Paradigms*, pages 33–39, 2001.