The Traust Authorization Service

ADAM J. LEE and MARIANNE WINSLETT
University of Illinois at Urbana-Champaign
and
JIM BASNEY and VON WELCH
National Center for Supercomputing Applications

In recent years, trust negotiation has been proposed as a novel authorization solution for use in *open-system* environments, in which resources are shared across organizational boundaries. Researchers have shown that trust negotiation is indeed a viable solution for these environments by developing a number of policy languages and strategies for trust negotiation that have desirable theoretical properties. Further, existing protocols, such as TLS, have been altered to interact with prototype trust negotiation systems, thereby illustrating the utility of trust negotiation. Unfortunately, modifying existing protocols is often a time-consuming and bureaucratic process that 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 trust negotiation systems. Traust acts as an authorization broker that issues access tokens for resources in an open system after entities use trust negotiation 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 trust negotiation 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.

This research was supported by the National Center for Supercomputing Applications; by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695; and by Sandia National Laboratories under grant DOE SNL 541065. Lee was also supported in part by a Motorola Center for Communications graduate fellowship.

A preliminary version of this paper appears under the title "Traust: A Trust Negotiation-Based Authorization Service for Open Systems" in the Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006).

Authors' addresses: A.J. Lee and M. Winslett, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801; emails: adamlee@cs.uiuc.edu, winslett@cs.uiuc.edu. J. Basney and V. Welch, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 1205 W. Clark St., Urbana, IL 61801; emails: jbasney@ncsa.uiuc.edu, vwelch@ncsa.uiuc.edu.

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

General Terms: Management, Security

Additional Key Words and Phrases: Attribute-based access control, credentials, trust negotiation

ACM Reference Format:

Lee, A.J., Winslett, M., Basney J., and Welch, V. 2008. The Traust authorization service. ACM Trans. Inform. Syst. Secur. 11, 1, Article 2 (February 2008), 33 pages. DOI: 10.1145/1330295.1330297. http://doi.acm.org/10.1145/1330295.1330297.

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 authorization decisions in these systems is a difficult task, as a potentially unbounded number of users and resources exists in an environment with few guarantees regarding preexisting trust relationships. Traditional authorization systems fail to work in these systems either because they cannot scale to such a large user base or because they make unrealistic assumptions about existing trust relationships in the system. As open systems continue to gain popularity, it is critical that the authorization 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, authorization 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 [Becker and Sewell 2004; Bertino et al. 2004; Bonatti and Samarati 2000; Herzberg et al. 2000; Hess et al. 2004; Koshutanski and Massacci 2004b; Li and Mitchell 2003; Wang et al. 2004; Winsborough et al. 2000; Yu et al. 2003]) and the embedding of trust negotiation into commonly used protocols [Hess et al. 2002]. 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 acts 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 [Bertino et al. 2004] or TrustBuilder [Winslett et al. 2002]) 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 article 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, comment on its performance, and discuss our experiences using Traust to broker access to existing legacy services. Section 7 illustrates the potential utility of our Traust client and server APIs by proposing several modifications to the GridFTP data transfer protocol that enable the tight integration of GridFTP with Traust. In Section 8 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 9.

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.

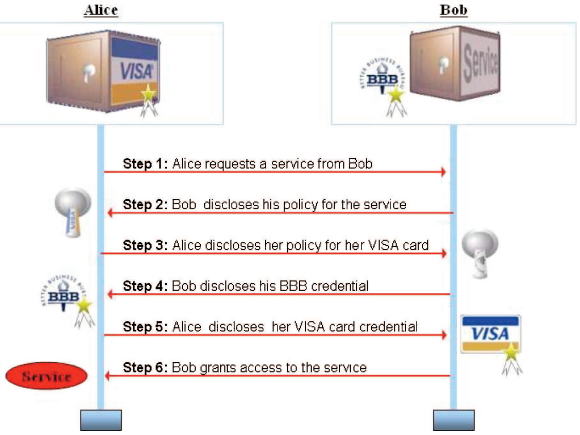

Alice
Bob
VISA
BBB
Service
Step 1: Alice requests a service from Bob
Step 2: Bob discloses his policy for the service
Step 3: Alice discloses her policy for her VISA card
Step 4: Bob discloses his BBB credential
Step 5: Alice discloses her VISA card credential
Step 6: Bob grants access to the service

Fig. 1. An example trust negotiation.

2.1 Trust Negotiation

Trust negotiation is a technique that has been proposed to address the scalability limitations of existing authorization 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 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.

Figure 1 shows an example trust negotiation in which a user, Alice, wishes to gain access to a service provided by Bob. After Alice requests access to Bob's service, Bob discloses the access policy for his service, which states that, to use the service, Alice must disclose a Visa cardholder credential. To protect herself from identity theft, Alice is willing to disclose this credential only to members of the Better Business Bureau (BBB), so rather than disclose her Visa cardholder credential, Alice sends Bob a release policy to this effect. Bob is in fact a member of the BBB and is willing to disclose this credential to anyone. This satisfies Alice, who discloses her Visa cardholder credential to Bob and is then granted access to Bob's service.

Authorization 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., Becker and Sewell [2004]; Bertino et al. [2003]; Herzberg et al. [2000]; Li and Mitchell [2003]), protocols and strategies for conducting trust negotiations or constructing distributed proofs (e.g., Bauer et al. [2005]; Bertino et al. [2004]; Koshutanski and Massacci [2004b, 2005]; Li et al. [2005]; Minami and Kotz [2005]; Winsborough and Li [2002]; Yu et al. [2003]), and logics for reasoning about the outcomes of these negotiations (e.g., Bonatti and Samarati [2000]; Winslett et al. [2005]). The foundational results presented in these works have also been shown to be viable authorization solutions for real-world systems through a series of implementations (such as those presented in Bauer et al. [2005]; Bertino et al. [2004]; Koshutanski and Massacci [2004a]; Minami and Kotz [2006]; Winslett et al. [2002]), 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 Hess et al. [2002]; ISRL [2005]). Unfortunately, trust negotiation is in many ways fundamentally different from previous authorization 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 authorization for the World Wide Web is presented by Hess and colleagues [2002]. 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 gain access to resources within their protection domains. Traust servers issue access tokens, encoded in formats understood by existing applications, to authorized users. 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 authorization 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 that 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 that should be satisfied by any authorization 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 Visa cardholder credential), we cannot assume that resource providers will know a priori the identities of all authorized clients who 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 have access to their resources, we can immediately recognize four important requirements that must be satisfied by any authorization system designed for use in open computing systems:

(1) *Bilateral trust establishment*. To enable effective resource sharing, we cannot require preexisting trust relationships between clients and resource providers; it is important to allow these entities to establish trust relationships with one another at runtime.
(2) *Runtime access policy discovery*. In large-scale open systems, clients cannot be expected to know a priori the access polices protecting resources of interest. Authorization systems used in these environments should allow clients to discover these policies as they become relevant.
(3) *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.
(4) *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 lightweight 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:

(5) *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

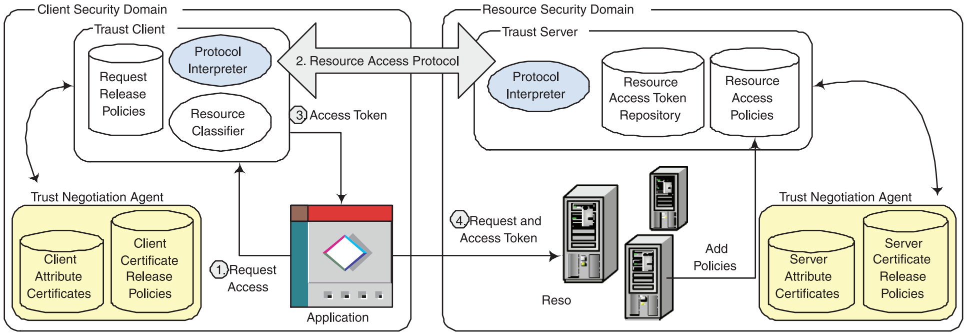


Fig. 2. Traust system architecture.

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 8.1, we revisit these requirements and discuss how Traust satisfies each of them.

4. TRAUST SYSTEM ARCHITECTURE

Figure 2 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 gain access to classified data), Traust clients may have a local resource classifier that can be used to determine the sensitivity classification (or classifications) of a particular resource request and identify its corresponding release policy (or policies). For maximum flexibility, a Traust client use can gain access to a directly 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 [Bertino et al. 2004], TTG [Winsborough and Li 2002], or TrustBuilder [Winslett et al. 2002] 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 have access to 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 gain access to 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 [Basney et al. 2005; Novotny et al. 2001] 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 Appendix.

5.1 Session Security

All communications between a client and Traust server occur inside a TLS [Dierks and Allen 1999] 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 metadata 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 be sent by the client only 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 for which the Traust client wishes to acquire access tokens. This naming system is flexible enough to 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 gain access to 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. On receiving an ETN message, the receiver will cease forwarding subsequent messages to its 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 with which the user wishes to negotiate for access. This description may be generated explicitly by the human user (e.g., after reading a Web page describing how to gain access to 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 3, the Traust client initiates zero or more content-triggered trust negotiation sessions [Hess et al. 2004] 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 to the Traust server describing the resource that the user wishes to access. This disclosure is indicated by the label B in Figure 3.

On 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,

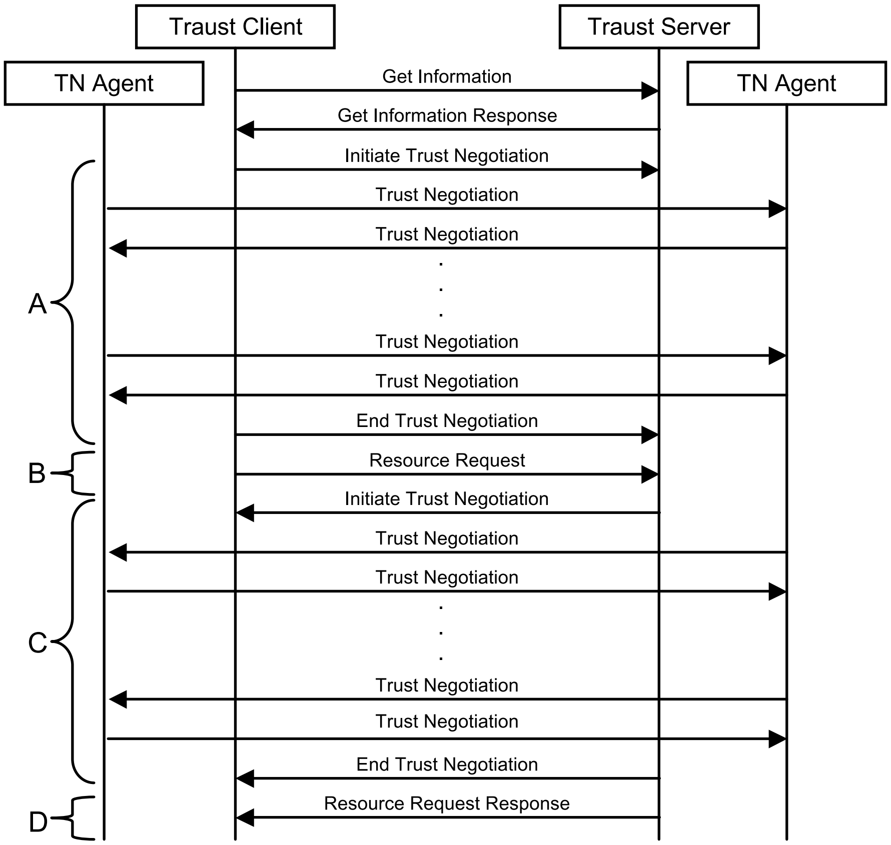


Fig. 3. The Traust resource access protocol.

indicated by the label C in Figure 3, 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 3, 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 [Pearlman et al. 2002]) to obtain the needed access tokens. To help

$$(GI_C\, GI_S)?(ITN_C(TN)^*ETN_C)^*RR_C(ITN_S(TN)^*ETN_S)?RR_S$$

Fig. 4. A regular expression describing successful executions of the resource access protocol.

detect certain types of attacks, it is important to differentiate between valid and invalid executions of the resource access protocol. Figure 4 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 on 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 that 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 that allows users to develop custom request sensitivity classifiers. Because defining these types of classifiers can be difficult, a user's organization (e.g., his or her employer or credential issuer) is likely to supply 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 that can be used to interface with a wide variety of access token repositories. We have developed a flexible token repository that allows the server to obtain the tokens needed to gain access to a given resource by either (1) gaining access to 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 use trust negotiation agents based on the TrustBuilder framework for trust negotiation [Winslett et al. 2002].

TrustBuilder currently supports the use of X.509 attribute certificates for credentials and the IBM Trust Policy Language [Herzberg et al. 2000] 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 [Hess et al. 2004, 2002; Ryutov et al. 2005], making it a good choice for use in the Traust system. We do note, however, that TrustBuilder does not currently support dynamic trust chain construction, in which mutually trusted third parties are discovered at runtime (e.g., as in Li et al. [2003]). This limits the applicability of our implementation to situations in which such trusted credential issuers are mutually agreed on by both parties at the start of the negotiation.

6.2 Usage with Legacy Resources

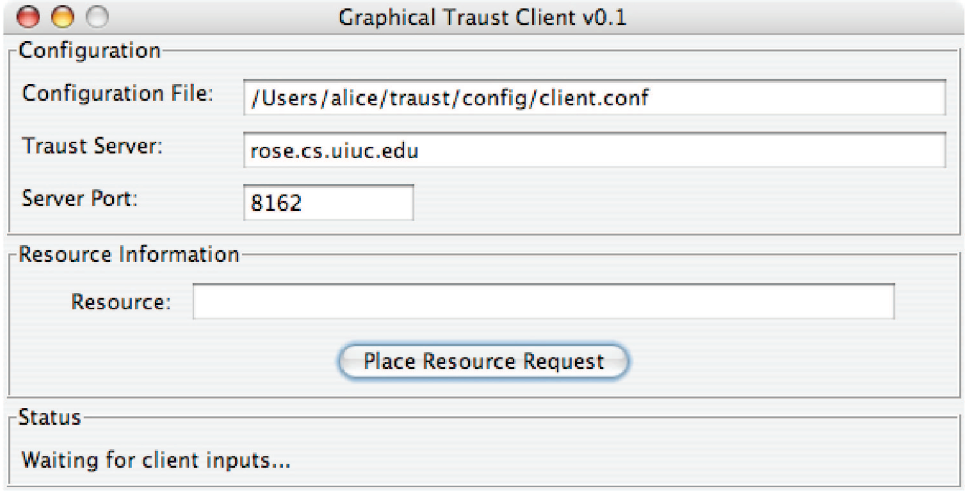
In some computing environments, it is considered acceptable to require that clients manually acquire resource access tokens prior to using networked services. For instance, at many universities, users wishing to have access to student records must first acquire a Kerberos ticket using a stand-alone client application (e.g., `kinit`). For these types of 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 5 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 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 have developed client and server APIs to enable 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.



Configuration File: /Users/alice/traust/config/client.conf

Traust Server: rose.cs.uiuc.edu

Server Port: 8162

Resource:

Waiting for client inputs...

Fig. 5. The graphical Traust client.

In Section 7, we demonstrate the potential utility of this type of integration by proposing two modifications to GridFTP, a secure mass data transfer protocol used heavily in the context of grid computing.

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 ten 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 his or her 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 willing to disclose his access request only to Traust servers that can prove that they are operated by a state-sponsored disaster response coordinator, and he 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.

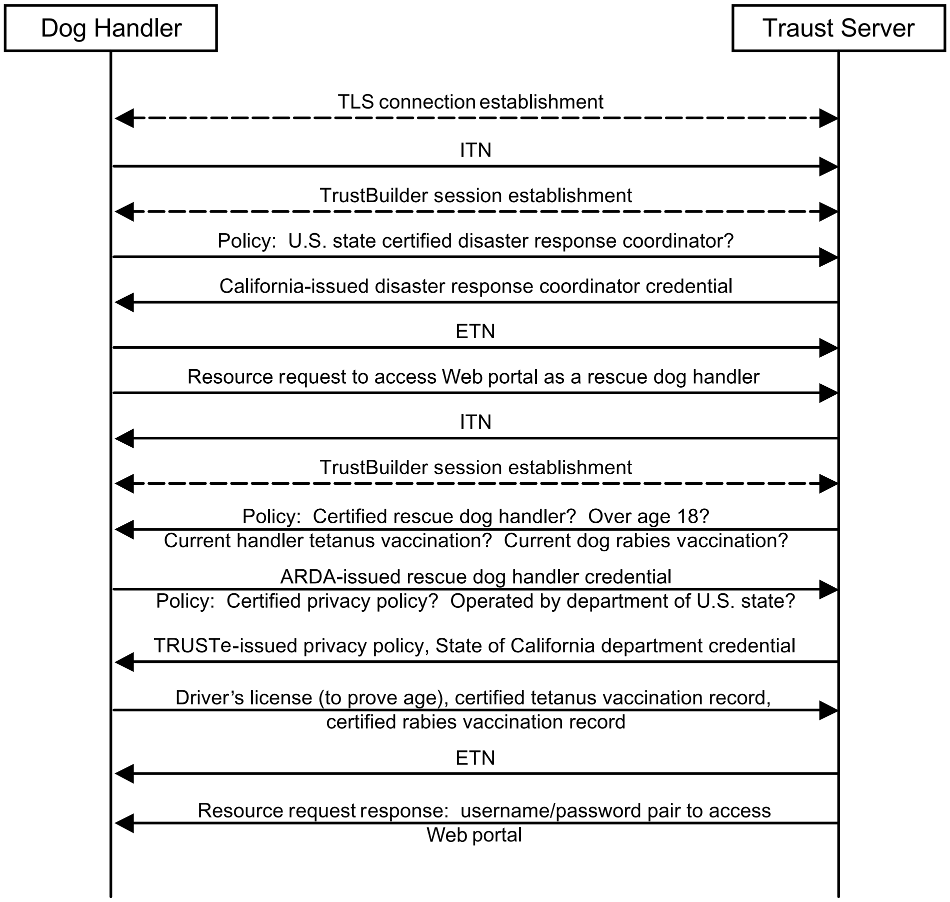

Dog Handler
Traust Server
TLS connection establishment
ITN
TrustBuilder session establishment
Policy: U.S. state certified disaster response coordinator?
California-issued disaster response coordinator credential
ETN
Resource request to access Web portal as a rescue dog handler
ITN
TrustBuilder session establishment
Policy: Certified rescue dog handler? Over age 18? Current handler tetanus vaccination? Current dog rabies vaccination?
ARDA-issued rescue dog handler credential
Policy: Certified privacy policy? Operated by department of U.S. state?
TRUSTe-issued privacy policy, State of California department credential
Driver's license (to prove age), certified tetanus vaccination record, certified rabies vaccination record
ETN
Resource request response: username/password pair to access Web portal

Fig. 6. Details of the rescue dog Traust scenario. Multiple-message exchanges with details omitted are denoted using dashed lines.

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. The details of this interaction are illustrated in Figure 6.

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 authorization systems cannot be used in open systems environments because 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 that takes 0.93 seconds on average; configuring the client to use a stand-alone trust negotiation agent would eliminate this overhead. Further, the TrustBuilder system is a prototype framework for trust negotiation that has not been optimized for performance. We have reason to believe that future research in trust negotiation frameworks will lead to trust negotiation agents with much better performance. In this case, 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, such a study would be of value once high-performance policy evaluators such as CPOL [Borders et al. 2005] are integrated with existing trust negotiation systems.

7. TRAUST INTEGRATION WITH GRIDFTP

In this section, we demonstrate one way in which the Traust service might be tightly coupled with a Traust-aware protocol. Specifically, we show how to extend the GridFTP protocol [Allcock 2003] to support dynamic access credential discovery via interaction with one or more Traust servers. Such tight integration allows users of the GridFTP server to obtain the credentials necessary to gain access to data provided by the server at runtime and eliminate the need for users to establish accounts on the server itself. Our primary design goal was to use the existing GSI authentication mechanisms [Welch et al. 2003] as much as possible to avoid unnecessary implementation costs.

7.1 The `TRAUST` Command

To facilitate the incorporation of Traust into GridFTP, we define a new FTP [Postel and Reynolds 1985] command, `TRAUST`. To initiate a GridFTP session supporting TRAUST interaction, the client opens an FTP connection to the GridFTP server and then issues the command:

```
TRAUST
```

This command should be entered prior to logging in to the system. The GridFTP server's response to this command will be response code 200. The body of this response will be as follows:

```
200-TRAUST <Traust server name or IP address>:<port>
200-<resource URI>
200-ATTRIB=(<attribute>,<value>)    // zero or more
200 <authentication method>
```

In this response (which is a valid FTP response, as per Postel and Reynolds [1985]), the port specification is optional and will be assumed to be 8162 when not present. The `<resource URI>` is a URI [Berners-Lee et al. 2005] that can be embedded in a Traust resource request command, and the optional `ATTRIB` lines further specify the resource. At this point, the client contacts the Traust server described in the first line of the GridFTP server's `200` response and requests access to the resource described on the second and third lines of the `200` response.

Should the exchange with the Traust server succeed, the Traust server will then issue the client the credentials needed to log into the GridFTP server. The client then uses the authentication method listed on the last line of the GridFTP server's response to the TRAUST command along with the newly obtained credentials to log into the server. Acceptable authentication method descriptions that can be embedded in the `200` response include "USERPASS" and "GSI," which represent the standard FTP username/password login and GSI authentication, respectively.

7.2 Enabling Permission Changes During a GridFTP Session

Although the `TRAUST` command is useful for determining whether a given client should be granted access to a particular GridFTP server, it will rarely be the case that all clients authorized to access a particular server should have access to all files stored on the server. To address this problem, we wish to allow clients the ability to negotiate for new access rights dynamically as they traverse the file system. This can be accomplished through the use of "access hints" embedded in GridFTP server error messages.

For instance, if we would like to enforce access lists on a per-directory basis, as in the Andrew file system [Morris et al. 1986], then the act of changing directories might cause authorization errors. A client who issues a `CWD` or `CDUP` command to the GridFTP server might be returned an error code `550` (access denied). This message usually takes the following form:

```
550 Access Denied
```

Rather than returning the standard FTP error code `550`, a GridFTP server supporting the `TRAUST` command returns the following access denied message with an embedded access hint:

```
550-TRAUST <Traust server name or IP address>:<port>
550-<resource URI>
```

```
500-ATTRIB=(<attribute>,<value>)    // zero or more
550-<authentication method>
550 Access Denied
```

The `<resource URI>`, optional `ATTRIB` lines, and `<authentication method>` fields above carry the same meaning as those returned in response to the `Traust` command; again, the port specification is optional. This enhanced error message will be interpreted by the client as an indicator that the current request failed, though a successful interaction with the indicated Traust server will lead to the access credentials necessary to allow the failed operation to succeed. In this case, the client will take the following actions:

(1) The client's Traust API is used to contact the indicated Traust server and request access to the specified resource.
(2) If the Traust interaction is successful, the client will be issued new access credentials by the Traust server that will allow access to the requested directory.
(3) The client then uses the newly obtained credentials in a reauthentication exchange with the GridFTP server initiated by using the GridFTP `AUTH` command. This exchange will take place using the method specified in the access hint provided to the client.
(4) The `CWD` or `CDUP` command can then be reissued and will succeed.

It is reasonable to assume that a good GridFTP client protocol interpreter will automate steps 1 through 4 after receiving an error message containing an access hint. In this case, the user of the GridFTP client would not need to manually intervene to trigger any of the interactions with the Traust server and would notice only a slight lag in the connection as the access credentials are renegotiated. To enforce the directory level access controls discussed in this section, servers will need to return error messages containing access hints not only with failed `CWD` and `CDUP` commands but also with failed commands that attempted to access files in directories other than the current working directory (e.g., commands such as `RETR`, `STOR`, and `LIST`).

One of the interesting features of this approach is that access hints can be embedded in any standard FTP error message. The client protocol interpreter need only parse error responses for the "`TRAUST`" string to know that a Traust interaction might possibly fix the error. This flexibility allows a GridFTP server to enable access hints that can enforce a wide variety of authorization policies on the server's resources. For instance, because the server is free to embed different access hints in `RETR` or `STOR` requests for the same file, this system can be used to enforce different read and write access controls for a single file or directory. This mechanism can, in turn, be used to enforce the principle of least privilege [Saltzer and Schroeder 1975].

7.3 An Example Traust-Enabled GridFTP Session

A sample Traust-enabled GridFTP session is shown in Figure 7. Alice first uses her GridFTP client to establish a connection to the server `GridFTP.foo.org`.

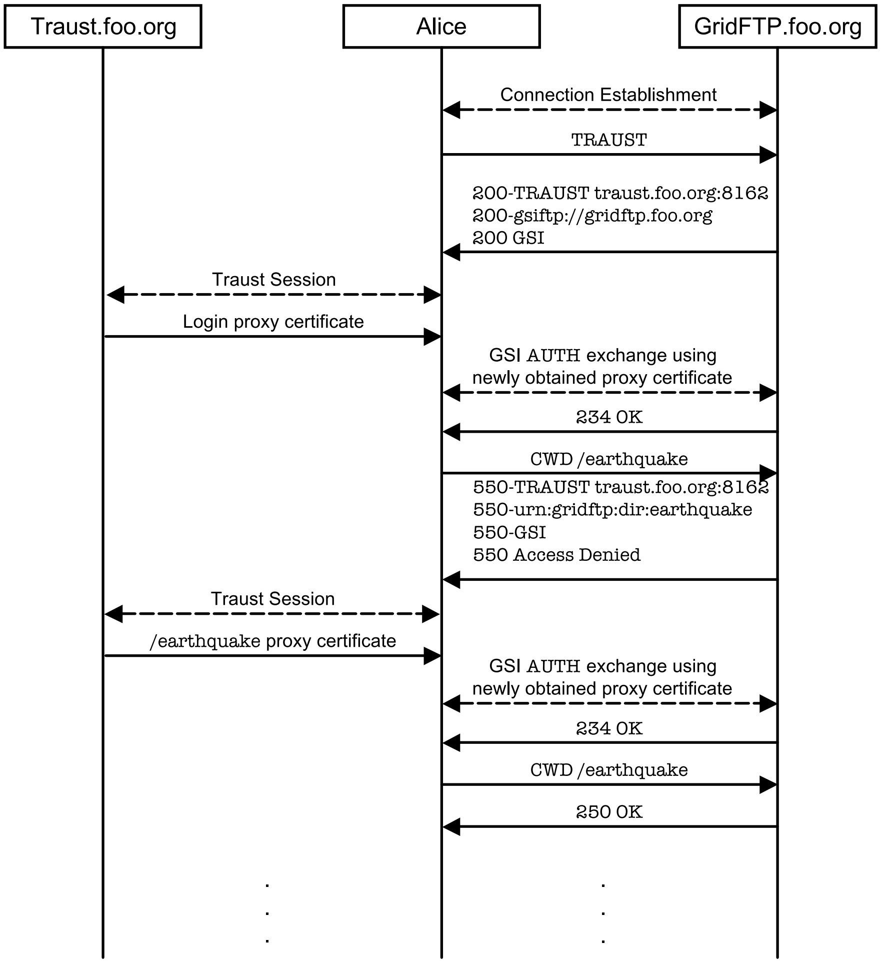


Fig. 7. A sample Traust-enabled GridFTP Session. Multiple-message exchanges with details omitted are denoted using dashed lines.

Because Alice has never had access to this server before, her GridFTP client issues the `TRAUST` command to determine the location of a Traust server that can be used to negotiation for an access certificate to this GridFTP server. The `200` response returned by the GridFTP server informs Alice's GridFTP client of the location of the Traust server used to broker access to the GridFTP server (`Traust.foo.org:8162`) and the name of the resource for the client to request access to in order to obtain login rights for the server (`gsiftp://gridftp.foo.org`). Alice's GridFTP client then uses the Traust client API to negotiate with the Traust server; the end result of this interaction is that the Traust server issues Alice's GridFTP client an X.509 proxy certificate permitting login to the designated GridFTP server. The GridFTP client then carries out a GSI `AUTH` exchange using this credential and is granted access to the server. Alice then attempts to gain access to the `/earthquake` directory on the server, but her access is denied. Her GridFTP client application parses the embedded access hint from the `550` error response, negotiates for permission to gain access to `/earthquake`, and reauthenticates to the server; this entire privilege enhancement process takes place without intervention from Alice. Alice can then continue her session with the GridFTP server.

The above example illustrates the flexibility that can be afforded by applications that interface natively with the Traust service. The introduction of the `TRAUST` command to the GridFTP protocol specification, along with a slight alteration of the semantics of error responses in certain circumstances, can provide a previously unknown client with enough information to dynamically acquire access tokens for the service at runtime without requiring active participation by the human user of the system. The syntax of a `550` response remains unchanged from that proposed by Postel and Reynolds [1985], thereby making these changes transparent to clients not supporting native Traust interaction. Although the relatively easy upgrade path demonstrated for GridFTP is unlikely to exist for all legacy applications, it nonetheless demonstrates that Traust can be coupled tightly with applications that wish to leverage its strengths. This type of compatibility is important to ensure not only that Traust serves as a short-term solution for introducing the strengths of trust negotiation to legacy environments but that it will remain a useful authorization model in the long run.

8. 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.

8.1 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.

8.2 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.

8.2.1 *Session Security.* In the Traust system, session security is provided through the use of the TLS protocol. Dierks and Allen [1999], present a security analysis of the TLS protocol under the assumption of an active attacker [Dolev and Yao 1983] 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 by Dierks and Allen [1999] 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 [Ylonen and Lonvick 2005] 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 noncertified 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 by Dierks and Allen [1999] ensures that messages exchanged during the Traust resource access protocol can be viewed only by their intended recipients.

8.2.2 *Ensuring Valid Attribute Certificates.* A naive implementation of the trust negotiation agents used in the Traust system leaves parties open to an attack in which a malicious party can demonstrate “proof” of ownership for attributes that it does not possess by conducting interleaved executions of the resource access protocol with multiple parties. We now describe the attack in greater detail and present a solution that eliminates the possibility of this attack from the Traust system.

To illustrate this attack, consider the following example, in which Alice is carrying out an execution of the Traust resource access protocol with Mallory. Alice wishes to gain access to Mallory's Music Warehouse so that she can download songs to listen to during her commute to work. During the client trust establishment stage of the resource access protocol, Mallory requests that Alice submit her digital Visa card (presumably so that Mallory can bill Alice for the music that she downloads) and demonstrate proof of ownership. To protect herself from identity theft, Alice is willing to disclose this credential only to businesses who are members of the Better Business Bureau (BBB). Alice sends this release policy to Mallory along with a challenge for him to sign using the private portion of his BBB credential. Mallory is not a member of the BBB, though he wishes to trick Alice into believing that he is.

Mallory then opens a Traust connection to Bob's Books. Bob runs a Traust server to allow certain groups of users, such as school teachers, to negotiate for access to discount coupons for the books that he sells. During the server trust establishment phase of the resource access protocol, Mallory informs Bob that he is willing to interact only with members of the Better Business Bureau and submits Bob a challenge (Alice's challenge!) to sign with the private portion of his BBB credential. Bob is a member of the BBB and willingly signs the challenge, returning it along with the public portion of his BBB credential to Mallory. Mallory then closes his connection with Bob and forwards Bob's BBB credential and signed challenge to Alice, who is now convinced that Mallory is a member of the BBB.

This attack is possible in the event that a naive implementation of the trust negotiation agent has no means of associating an attribute credential with a particular identity. That is, this attack is enabled not by Traust but by the trust negotiation agent used by Traust. This attack can be prevented in practice, however, if parties form a loose notion of "session identity" by requiring a binding between attribute credentials and the public key used to establish the underlying TLS tunnel. To prevent the attack discussed above, we can require that parties not sign the challenge sent by the other negotiating party directly. Rather, the trust negotiation agent should first concatenate this challenge with a hash of the public key that he or she used to establish the TLS tunnel used in this interaction and then sign the resulting bit string. This prevents the surreptitious forwarding of attribute credentials, as in the above example, as the remote party can verify that the challenge was concatenated with the expected public key hash prior to signing.

Note that the solution above does not prevent the collusion of malicious entities who wish to pool their resources to appear as a single, more privileged entity. Techniques such as hidden credentials and oblivious signature-based envelopes [Holt et al. 2003; Li et al. 2003] can be used to prevent collusion, though this system requires that parties know the identity used by the other negotiating party to obtain their identity-based public keys. Another possible solution involves the use of attribute-based encryption [Sahai and Waters 2005], though this requires that all attributes of interest to the negotiation be issued by the same authority. Unfortunately, this makes attribute-based encryption unappealing in the open system environment for which Traust was designed. Systems such as *idemix* [Camenisch and Herreweghen 2002] can also be used to prevent this type of collusion, though at the expense of embedding a master secret into each credential. The difficulty of solving the collusion-resistance problem makes it an exciting area for future work.

8.2.3 *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 on which the Traust server is running grants the attacker the access tokens needed to gain access to only 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. Such a redirect mechanism is not currently supported but could easily be added to the Traust protocol by defining a new credential encoding (see Appendix C) that encodes the next server to contact, rather than an access token. 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 [Blakley 1979; Shamir 1979]. 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. On 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, Novotny and colleagues [2001], 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 [O'Donnell and Sethu 2004; Zhang et al. 2001] to reduce 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.

8.2.4 *Denial of Service.* Even as optimized trust negotiation agents are developed and thus the performance of the Traust authorization service improves, the potentially centralized nature of a Traust deployment along with the relatively heavyweight process of trust negotiation 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 and colleagues [2005] integrate TrustBuilder with the GAA-API and leverage the GAA-API's mechanisms for responding to changes in system context. 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.

Another more standard technique to help reduce the risk of denial of service attacks on Traust servers is replication. Because 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 gain access to resources provided by the organization.

8.2.5 *User Accountability.* Another interesting problem related to systems based on trust negotiation involves the trade-offs that exist between openness and accountability. If a resource protected by an identity-based authorization system is compromised or attacked, the user whose identity was used to attack the system can be held accountable or at the very least investigated for clues as to who the real attacker might have been. Because trust negotiation is based on the exchange of attribute—rather than identity—information, this is not always possible.

As this problem is present in attribute-based trust negotiation systems in general, rather then Traust specifically, we did not address it during our system design. However, we see several possible solutions to this problem. One solution would involve the use of third-party pseudonymity certifiers. These entities could be used to provide a legal means of recourse for resource providers whose resources were abused by a user with a pseudonym attribute issued by that certifier. It is not clear how such a system would function in truly open environments, though it is an interesting avenue for future research. Alternatively, resource providers could use virtual fingerprinting techniques [Lee and Winslett 2006] to uniquely identify users who have abused their resources. These users could then be proactively blocked at other partner sites or perhaps even have their civil identities uncovered during the abuse investigation.

9. CONCLUSIONS AND 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 authorization 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.

One interesting direction for future work involves examining how Traust could 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, it is also important to further explore ways to securely manage the access tokens stored in the repository of a Traust server.

APPENDIX

In this appendix, we describe the wire protocol used by the Traust system. In particular, we discuss the message syntax omitted in Section 5, the means through which certain types of access tokens are encoded by the Traust system, and, in detail, an example Traust interaction.

A. CORE PROTOCOL

A.1 TCP Port Number

The Traust server can be configured to listen on any open TCP port. The default port is 8162.

A.2 Traust Session Transport Layer

Traust uses TLS as its underlying transport layer. As such, all communications discussed in this specification take place confidentially between the client and the service in a tamper-proof manner.

A.3 Basic Message Format

The basic message format used by Traust borrows greatly from that of the MyProxy system [Basney 2005]. Commands are written as lines of ASCII text, each of which terminates with the ASCII newline character, '\n' (0x0a). A command is terminated by two newline characters. Basic commands have the form:

```
COMMAND=<integer>
<command_body_line_1>
      .
      .
<command_body_line_n>
```

Responses to these commands can take one of two forms, a successful response or a failure response. Successful responses have the format:

```
COMMAND=<integer>
RESPONSE=0
<response_line_1>
      .
      .
<response_line_n>
```

Failure responses have the following format:

```
COMMAND=<integer>
```

```
RESPONSE=1
ERROR=<text_line_1>
      .
      .
ERROR=<text_line_n>
```

It is important to note that in both the successful and failure response cases the `<integer>` in the response `COMMAND` line is the same as the `<integer>` in the `COMMAND` line of the command that generated this response. In the event that the response returned is an error response (`RESPONSE=1`), the `ERROR` lines should be concatenated together (separated by '\n' characters) by the client process before presentation to the user. After sending an `ERROR` response, the server must close the communication channel that it shares with the client immediately.

B. CURRENTLY SUPPORTED MESSAGES

In this section, we discuss the commands supported by our prototype version of Traust (version 0.1).

B.1 Get Info Command (COMMAND=0)

The get info command is used by a client to obtain various metainformation regarding the Traust server. To request this information, the client sends the following command:

```
COMMAND=0
```

The server's response to this command is as follows:

```
COMMAND=0
RESPONSE=0
ATTRIB=(VERSION,0.1)
ATTRIB=(<attrib>,<value>)  \
      .                     \ optional
      .                     /
ATTRIB=(<attrib>,<value>)  /
```

In this response, the optional lines that begin with the `ATTRIB` prefix can be used to return various other information regarding the Traust server. Clients should be able to parse `ATTRIB=(CONTACT,(<name>,<email>))` and `ATTRIB=(MOTD,<msg>) ATTRIBs`. These `ATTRIBs` disclose the contact point for questions about this server and the server "message of the day," respectively. As with `ERROR` responses, multiple `MOTD` lines should be concatenated by the client (separated by '\n' characters) before presentation to the user. Servers may include site-specific `ATTRIBs` in their get info responses, though no assumptions should be made as to whether clients will know how to interpret these additional `ATTRIBs`.

B.2 Initiate Trust Negotiation (COMMAND=1)

The initiate trust negotiation command is sent to indicate that the sending party wishes to conduct a trust negotiation with the other party. Clients may send this command prior to disclosing a resource request (see Section B.4) to establish trust in the Traust server before disclosing a sensitive request. Servers may send this command after receiving a resource request command from the client, to establish trust in the client prior to disclosing access credentials for the requested resource.

On the receipt of this command, the receiving party should use its trust negotiation agent to parse all subsequent incoming communication until such time as it receives an end trust negotiation command. An initiate trust negotiation command has the following format:

```
COMMAND=1
```

As this command is an indicator, the body is left blank. It should also be noted that there is no "response" version of this command, as it serves simply as an indicator to the other negotiating party and does not require any response on its behalf.

B.3 End Trust Negotiation (COMMAND=2)

The end trust negotiation command is sent after the trust negotiation initiated by an initiate trust negotiation message has ended. At this point, the receiving party will cease to use its trust negotiation agent to parse incoming communication. An end trust negotiation command has the following format:

```
COMMAND=2
```

As with the initiate trust negotiation command, the body of the end trust negotiation command is left blank, and there is no "response" version of this command.

B.4 Resource Request (COMMAND=3)

The resource request command invokes the credential lookup functionality of the server. With this command, clients indicate a desire to access a given resource. A resource request has the following format:

```
COMMAND=3
<resource URI>
ATTRIB=(<attribute>,<value>)  // zero or more
```

The body of this command is a URI [Berners-Lee et al. 2005] identifying the resource to which the client wishes to have access. In many cases, this URI will take the form of a URL [Berners-Lee et al. 1994] specifying a networked service, though it may also take the form of a URN [Moats 1997] describing a more generic resource (such as a client's desire to activate a sitewide role). The URN syntax is also convenient for servers wishing to attach opaque identifiers to Traust resources in hopes of hindering information gathering attacks. The optional ATTRIB lines allow a resource request to contain additional information, in the form of ⟨attribute,value⟩ pairs.

If the server receives an invalid resource request (for example, a request for a nonexistent resource), it should return an error of the following form:

```
COMMAND=3
RESPONSE=1
ERROR=Invalid request
```

On receiving a valid resource request, the server may optionally send an initiate trust negotiation command and conduct a trust negotiation session to determine whether the client is authorized to have access to the given resource. Should the trust negotiation session fail to allow the server to establish trust in the client, the server should respond with an error response to the resource request. This message has the format:

```
COMMAND=3
RESPONSE=1
ERROR=Client not authorized
```

If the trust negotiation session allows the server to establish trust in the client (or was not required), the server will return the access credential (or credentials) needed to gain access to the requested resource by embedding them in a resource request response. These messages have the following format:

```
COMMAND=3
RESPONSE=0
BEGIN_CREDENTIAL       \
TYPE=<integer>          \
<credential_data>        \
      .                   > Zero or more
      .                  /
<credential_data>       /
END_CREDENTIAL         /
```

As shown above, zero or more credentials can be disclosed in each resource request response. Each such credential occurs between a matched pair of BEGIN_CREDENTIAL and END_CREDENTIAL lines. The TYPE line indicates (by way of an integer code) the type of credential contained in the immediately following CRED lines. The format of the CRED lines is TYPE dependent and is discussed in Section C of this appendix.

C. SUPPORTED CREDENTIAL TYPES

This section describes the minimum set of credential types that must be supported by any Traust server whose version is reported as 0.1.

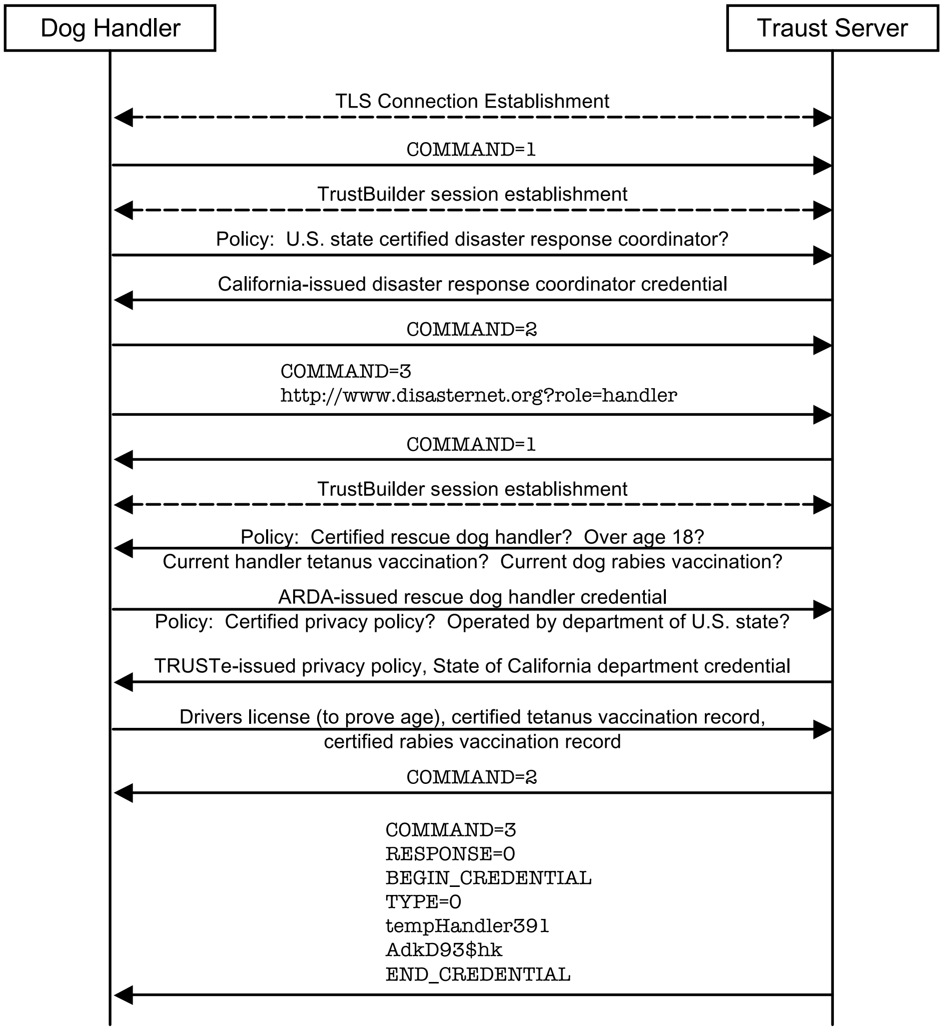

Dog Handler
Traust Server
TLS Connection Establishment
COMMAND=1
TrustBuilder session establishment
Policy: U.S. state certified disaster response coordinator?
California-issued disaster response coordinator credential
COMMAND=2
COMMAND=3
http://www.disasternet.org?role=handler
COMMAND=1
TrustBuilder session establishment
Policy: Certified rescue dog handler? Over age 18?
Current handler tetanus vaccination? Current dog rabies vaccination?
ARDA-issued rescue dog handler credential
Policy: Certified privacy policy? Operated by department of U.S. state?
TRUSTe-issued privacy policy, State of California department credential
Drivers license (to prove age), certified tetanus vaccination record,
certified rabies vaccination record
COMMAND=2
COMMAND=3
RESPONSE=0
BEGIN_CREDENTIAL
TYPE=0
tempHandler391
AdkD93$hk
END_CREDENTIAL

Fig. 8. Wire-level details of the rescue dog Traust scenario. Multiple-message exchanges with details omitted are denoted using dashed lines.

C.1 Username/Password Credentials (TYPE=0)

Username and password credentials will be transmitted using the following format:

```
BEGIN_CREDENTIAL
TYPE=0
<plaintext username>
<plaintext password>
END_CREDENTIAL
```

It should be noted that plaintext username/password disclosure is permissible, as the TLS channel that exists between the client and Traust server provides confidentiality and integrity for all data transmitted.

C.2 X.509 Proxy Certificates (TYPE=1)

We now define the message format for transmitting an X.509 proxy credential consisting of a proxy certificate [Tuecke et al. 2004], an RSA private key, and a supporting certificate chain. Such credentials will be encoded and transmitted in the following format:

```
BEGIN_CREDENTIAL
TYPE=1
<first line of the PEM encoded proxy certificate>
      .
      .
<Nth line of the PEM encoded proxy certificate>
END_CREDENTIAL
```

For example, if the proxy certificate being transmitted was generated using the `grid-proxy-init` command, the lines between `TYPE` and `END_CREDENTIAL` should contain, line for line, the contents of the `/tmp/x509up_u<uid>` file generated as the output from `grid-proxy-init`.

D. A SAMPLE TRAUST SESSION IN DETAIL

Figure 8 illustrates the wire-level details of the rescue dog Traust scenario discussed in Section 6.4. The textual description of this interaction is omitted because it is the same as that given in Section 6.4. As in the previous discussion of this scenario, the TrustBuilder interactions are left at a high level, as a more detailed treatment of this protocol is out of the scope of this paper.

REFERENCES

ALLCOCK, W. 2003. GridFTP protocol specification. Global Grid Forum Recommendation GFD.20. ⟨http://www.globus.org/alliance/publications/papers/GFD-R.0201.pdf⟩.

BASNEY, J. 2005. MyProxy protocol. Global Grid Forum Experimental Document GFD-E.54.

BASNEY, J., HUMPHREY, M., AND WELCH, V. 2005. The MyProxy online credential repository. *Soft.: Prac. Exper. 35,* 9 (July) 801–816.

BAUER, L., GARRISS, S., AND REITER, M. K. 2005. Distributed proving in access-control systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. 81–95.

BECKER, M. Y. AND SEWELL, P. 2004. Cassandra: Distributed access control policies with tunable expressiveness. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*. 159–168.

BERNERS-LEE, T., FIELDING, R. T., AND MASINTER, L. 2005. Uniform resource identifier (URI): Generic syntax. IETF Request for Comments RFC-3986.

BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. 1994. Uniform resource locators (URL). IETF Request for Comments RFC-1738.

BERTINO, E., FERRARI, E., AND SQUICCIARINI, A. C. 2003. 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)*. 81–84.

BERTINO, E., FERRARI, E., AND SQUICCIARINI, A. C. 2004. Trust-X: A peer-to-peer framework for trust establishment. *IEEE Trans. Knowl. Data Eng. 16,* 7 (July) 827–842.

BLAKLEY, G. R. 1979. Safeguarding cryptographic keys. In *AFIPS Conference Proceedings*. Vol. 48. 313–317.

BONATTI, P. AND SAMARATI, P. 2000. Regulating service access and information release on the Web. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*. 134–143.

BORDERS, K., ZHAO, X., AND PRAKASH, A. 2005. CPOL: High-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 147–157.

CAMENISCH, J. AND HERREWEGHEN, E. V. 2002. Design and implementation of the *idemix* anonymous credential system. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 21–30.

DIERKS, T. AND ALLEN, C. 1999. The TLS protocol version 1.0. IETF Request for Comments RFC-2246.

DOLEV, D. AND YAO, A. C. 1983. On the security of public key protocols. *IEEE Trans. Inform. Theory IT-29,* 2 (March) 198–208.

HERZBERG, A., MASS, Y., MICHAELI, J., NAOR, D., AND RAVID, Y. 2000. Access control meets public key infrastructure, or: assigning roles to strangers. In *Proceedings of the IEEE Symposium on Security and Privacy*. 2–14.

HESS, A., HOLT, J., JACOBSON, J., AND SEAMONS, K. E. 2004. Content-triggered trust negotiation. *ACM Trans. Inform. Syst. Secur. 7,* 3 (Aug.) 428–456.

HESS, A., JACOBSON, J., MILLS, H., WAMSLEY, R., SEAMONS, K. E., AND SMITH, B. 2002. Advanced client/server authentication in TLS. In *Proceedings of the Network and Distributed Systems Security Symposium*. 203–214.

HOLT, J., BRADSHAW, R., SEAMONS, K. E., AND ORMAN, H. 2003. Hidden credentials. In *Proceedings of the 2nd ACM Workshop on Privacy in the Electronic Society*. 1–8.

ISRL. 2005. Internet security research lab–projects. ⟨`http://isrl.cs.byu.edu/TrustBuilder.html`⟩.

KOSHUTANSKI, H. AND MASSACCI, F. 2004a. Interactive access control for web services. In *Proceedings of the 19th IFIP Information Security Conference (SEC)*. 151–166.

KOSHUTANSKI, H. AND MASSACCI, F. 2004b. Interactive trust management and negotiation scheme. In *Proceedings of the Second International Workshop on Formal Aspects in Security and Trust (FAST)*. 139–152.

KOSHUTANSKI, H. AND MASSACCI, F. 2005. Interactive credential negotiation for stateful business processes. In *Proceedings of the 3rd International Conference on Trust Management (iTrust)*. 257–273.

LEE, A. J. AND WINSLETT, M. 2006. Virtual fingerprinting as a foundation for reputation in open systems. In *Proceddings of the 4th International Conference on Trust Management (iTrust'06)*. Number 3986 in Lecture Notes in Computer Science. Springer, 236–251.

LI, J., LI, N., AND WINSBOROUGH, W. H. 2005. Automated trust negotiation using cryptographic credentials. In *Proceedings of 12th ACM Conference on Computer and Communications Security (CCS)*. 46–57.

LI, N., DU, W., AND BONEH, D. 2003. Oblivious signature-based envelope. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*. 182–189.

LI, N. AND MITCHELL, J. C. 2003. RT: A role-based trust-management framework. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*. 201–212.

LI, N., WINSBOROUGH, W. H., AND MITCHELL, J. C. 2003. Distributed credential chain discovery in trust management. *J. Comput. Secur. 11,* 1 (Feb.) 35–86.

MINAMI, K. AND KOTZ, D. 2005. Secure context-sensitive authorization. *J. Pervas. Mobile Comput. 1,* 1 (March) 123–156.

MINAMI, K. AND KOTZ, D. 2006. Scalability in a secure distributed proof system. In *Proceedings of the International Conference on Pervasive Computing*. 220–237.

MOATS, R. 1997. URN syntax. IETF Request for Comments RFC-2141.

MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S., AND SMITH, F. D. 1986. Andrew: A distributed personal computing environment. *Comm. ACM 29,* 3 (March) 184–201.

NOVOTNY, J., TUECKE, S., AND WELCH, V. 2001. An online credential repository for the grid: MyProxy. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC-10)*. 104–111.

O'DONNELL, A. J. AND SETHU, H. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. 121–131.

PEARLMAN, L., WELCH, V., FOSTER, I., KESSELMAN, C., AND TUECKE, C. 2002. A community authorization service for group collaboration. In *Proceedings of the 3rd IEEE International Workshop on Policies for Distributed Systems and Networks*. 50–59.

POSTEL, J. AND REYNOLDS, J. 1985. File transfer protocol (FTP). IETF Request for Comments RFC-959.

RYUTOV, T., ZHOU, L., NEUMAN, C., LEITHEAD, T., AND SEAMONS, K. E. 2005. Adaptive trust negotiation and access control. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*. 139–146.

SAHAI, A. AND WATERS, B. 2005. Fuzzy identity based encryption. In *Proceedings of Eurocrypt*. Lecture Notes in Computer Science, vol. 3494, Springer, 457–473.

SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. In *Proceedings of IEEE 63,* 9 (Sept.) 1278–1308.

SHAMIR, A. 1979. How to share a secret. *Comm. ACM 22,* 11 (Nov.) 612–613.

TUECKE, S., WELCH, V., ENGERT, D., PEARLMAN, L., AND THOMPSON, M. 2004. Internet X.509 public key infrastructure (PKI) proxy certificate profile. IETF Request for Comments RFC-3820.

WANG, L., WIJESEKERA, D., AND JAJODIA, S. 2004. A logic-based framework for attribute based access control. In *Proceedings of the 2nd ACM Workshop on Formal Methods in Security Engineering (FMSE'04)*. 45–55.

WELCH, V., SIEBENLIST, F., FOSTER, I., BRESNAHAN, J., CZAJKOWSKI, K., GAWOR, J., KESSELMAN, C., MEDER, S., PEARLMAN, L., AND TUECKE, S. 2003. Security for grid services. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC12)*. 48–57.

WINSBOROUGH, W. H. AND LI, N. 2002. Towards practical automated trust negotiation. In *Proceedings of the 3rd IEEE International Workshop on Policies for Distributed Systems and Networks*. 92–103.

WINSBOROUGH, W. H., SEAMONS, K. E., AND JONES, V. E. 2000. Automated trust negotiation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*. 88–102.

WINSLETT, M., YU, T., SEAMONS, K. E., HESS, A., JACOBSON, J., JARVIS, R., SMITH, B., AND YU, L. 2002. The TrustBuilder architecture for trust negotiation. *IEEE Intern. Comput. 6,* 6 (Nov./Dec.) 30–37.

WINSLETT, M., ZHANG, C., AND BONATTI, P. A. 2005. PeerAccess: A logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 168–179.

YLONEN, T. AND LONVICK, C. 2005. SSH transport layer protocol. IETF Network Working Group Internet-Draft.

YU, T., WINSLETT, M., AND SEAMONS, K. E. 2003. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Trans. Inform. Syst. Secur. 6,* 1 (Feb.) 1–42.

ZHANG, Y., VIN, H., ALVISI, L., LEE, W., AND DAO, S. K. 2001. Heterogeneous networking: A new survivability paradigm. In *Proceedings of the Workshop on New Security Paradigms*. 33–39.

Received October 2006; revised March 2007; accepted June 2007