PRESERVING TRUST ACROSS MULTIPLE SESSIONS IN OPEN SYSTEMS

by

Fuk Wing Thomas Chan

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

June 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Fuk Wing Thomas Chan

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Kent E. Seamons, Chair
Date	Quinn Snell
Date	Bryan S. Morse

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Fuk Wing Thomas Chan in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Kent E. Seamons Chair, Graduate Committee

Accepted for the Department

Tony R. Martinez
Department Chair

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

PRESERVING TRUST ACROSS MULTIPLE SESSIONS IN OPEN SYSTEMS

Fuk Wing Thomas Chan

Department of Computer Science

Master of Science

Trust negotiation, a new authentication paradigm, enables strangers on the Internet to establish trust through the gradual disclosure of digital credentials and access control policies. Previous research in trust negotiation does not address issues in preserving trust across multiple sessions. This thesis discusses issues in preserving trust between parties who were previously considered strangers. It also describes the design and implementation of trust preservation in TrustBuilder, a prototype trust negotiation system. Preserving trust information can reduce the frequency and cost of renegotiation. A scenario is presented that demonstrates that a server supporting trust preservation can recoup the cost of the trust preservation facility when approximately 25% of its requests are from repeat customers. The throughput and response time improve up to approximately 33% as the percentage of repeat customers grows to 100%.

ACKNOWLEDGMENTS

First, I appreciate your interest in reading this work. You make this work valuable. I also appreciate my committee members Dr. Snell and Dr. Morse; my fellow research assistants in the Internet Security Research Lab, Travis, Evan, Robert, Lina, Jared, Adam, Ryan, Jason, Tore, Bryan, Tim, Jim, Nate and many others. Thank you for allowing me to bounce off ideas and for providing me feedback on various chapters. You made this work interesting and fun. My thanks also go to Dr. Knutson who provided valuable input on my spring research presentations which became part of this thesis.

I would also like to express my thanks to individuals who reached down to lift me up and helped me feel hopeful: Dr. Cory Barker who invited me to his CS252, my first formal computer science class. Through his devotion and interest, I was converted to computer science. Dr. Christopher Jones, who was brave enough to let me present his paper in front of more than thirty PhDs at an international conference. It is he who sparked my interest in research and to continue my schooling. My adviser, Dr. Kent Seamons, who saw my potential, inspired my interest in trust negotiation and supported me throughout this work. Through his genuine encouragement and refinement, I completed this work.

Most of all, I thank my Father in Heaven who sent me here to learn and grow. I especially thank His wisdom for providing me the opportunity to meet my wife Adeline. My dear Adeline, you are my guiding light. You gave me courage throughout my education. You renewed my energy, strengths and gave me the reasons to carry on. Thank you for taking care of our children, Himhim, Waiwai, and Yanyan while I camped at the lab. Thank you for all the rides and endless waiting in the parking

lot.

Standing with you side by side, my dreams never die.

Mountains are high; with you I can touch the sky.

This work was long, time goes by and by.

Abandon the work and say goodbye?

Oh no! With your courage,

Your advice,

I survive.

Financial support for this research was provided by DARPA through AFRL contract number F33615-01-C-0336 and through Space and Naval Warfare Systems Center San Diego grant number N66001-01-1-8908. Support was also provided by the National Science Foundation under grant number CCR-0325951 and prime cooperative agreement number IIS-0331707, and The Regents of the University of California.

Contents

1 Introduction **1**

1.1 Overview of trust negotiation 1

1.1.1 Trust negotiation example 1

1.1.2 Sensitive credentials 2

1.1.3 Sensitive policies 3

1.2 Motivation 3

1.3 Research goals 5

1.4 Thesis statement 5

1.5 Outline 7

2 Trust Preservation using a TRT **9**

2.1 Design goals 9

2.2 Trust resumption token 14

2.2.1 TRT example 15

2.2.2 TRT usage in a federation 18

3 Trust Preservation using Kerberos and KeyNote **21**

3.1 Kerberos integration 21

3.2 KeyNote integration 24

4 Performance Analysis **31**

4.1 Experimental framework 31

4.2 Results: throughput and response time 36

5 Related Work **41**

5.1 Cookies 41

5.2 Kerberos tickets 42

5.3 Trust negotiation in session level protocols 42

5.4 Single sign-on 44

6 Conclusion and Future Work **45**

List of Tables

4.1 Client and server specifications 32

List of Figures

1.1 A simplified trust negotiation. 2

2.1 An extended example to illustrate the usage of an authenticator . . . 10

2.2 The trust negotiation flow between Alice and Bob. 16

2.3 The trust negotiation flow between Alice and Bob with multiple rounds of negotiation. 16

2.4 Alice's TRT 17

2.5 Alice resumes the trust level using TRT. 18

3.1 The Kerberos distributed authentication system architecture. 22

3.2 A simple design for integrating TrustBuilder with Kerberos. 23

3.3 Applications use KeyNote to determine whether a client is authorized for a requested service. 24

3.4 A spending application using KeyNote to determine a user's spending power. 26

3.5 A typical flow of interaction that illustrates an application making a query to KeyNote based on a user's request. 27

3.6 The KeyNote API. 29

4.1 Various sized groups of users simultaneously negotiating trust for a protected resource. 33

4.2 A pharmacy scenario involving a three-round trust negotiation process. 35

4.3 Server throughput during trust negotiations involving a TRT. 37

4.4 Mean response time for trust negotiations involving a TRT. 37

4.5 An estimate of the cumulative cost of a trust negotiation while varying the number of users, number of requests per user, and the percentage of repeat customers. 40

Chapter 1 — Introduction

1.1 Overview of trust negotiation

Trust negotiation, a new authentication paradigm, enables strangers on the Internet to establish trust or proof of qualification through the gradual disclosure of credentials and access control policies [4, 15, 16]. In open systems like the Internet, the majority of transactions are between strangers—entities that are not in the same security domain and do not have a pre-existing relationship. The identity of the participants is often irrelevant, unknown, or must remain private. Under such circumstances, identity-based authentication is not suitable. Trust negotiation is designed to allow strangers to authenticate one another based on attributes contained in digital credentials.

Digital credentials are unforgeable and verifiable. The attributes embedded in a credential are asserted and signed by a trusted third party known as a certificate authority (CA). A common format for a digital credential is an X.509v3 certificate.

1.1.1 Trust negotiation example

The following is a simple example of trust negotiation. Alice has an accident while she is hiking out-of-state. A rescue team air lifts her to an emergency room (ER). Before Doctor Bob, the on-call anesthesiologist, prescribes any pain medication to Alice, he needs to find out whether Alice is allergic to any medication. The Health Insurance Portability and Accountability Act of 1996 (HIPAA) prohibits Alice's health care provider, Jones Clinic (JC), from releasing her medical records except to a qualified physician who is providing treatment to her.[1] Doctor Bob possesses an anesthesiologist credential that is signed by the American Medical Association

[1]This oversimplified example conveys the general idea of HIPAA.

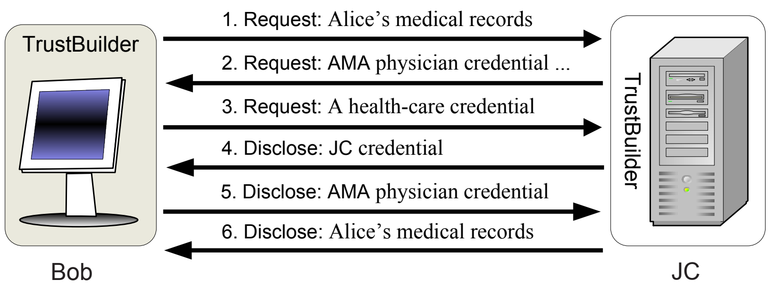

TrustBuilder
Bob
1. Request: Alice's medical records
2. Request: AMA physician credential ...
3. Request: A health-care credential
4. Disclose: JC credential
5. Disclose: AMA physician credential
6. Disclose: Alice's medical records
TrustBuilder
JC

Figure 1.1: A simplified trust negotiation.

(AMA). Doctor Bob only discloses his credential to other physicians and health care organizations to avoid unwanted solicitation from salespersons. Jones Clinic possesses a credential stating that it is a registered and licensed health care facility under the state department of health.

Figure 1.1 depicts the trust negotiation process between Doctor Bob and JC. Doctor Bob's client software requests Alice's medical records from JC's server. JC's server requests a credential proving that Doctor Bob is a qualified physician. Bob's client requests a credential proving that JC is a physician or a health-care organization. Since the JC's health care credential is freely available upon request, JC's server sends the credential to Bob's client. After verifying the credential, Bob's client releases his anesthesiologist credential. JC's server examines and verifies the credential, and then releases Alice's health care records to Bob.

1.1.2 Sensitive credentials

The trust negotiation process is analogous to how strangers can build credibility or establish mutual trust through exchanging paper credentials. In trust negotiation, digital credentials are gradually disclosed and exchanged. It is common for people to hesitate to disclose credentials that contain their sensitive information to strangers. Similar to paper credentials, digital credentials may contain sensitive information (attributes) like birth date, credit card number, personal health information, social security number, or weight. When attributes in a digital credential are sensitive, an access control policy can control its disclosure. In the example above, Doctor Bob protected his anesthesiologist credential with an access control policy allowing only physicians or health-care organizations to access his credential.

1.1.3 Sensitive policies

Doctor Bob's access control policy governs the disclosure of his physician credential. The policy defines the attributes a requester must possess in order to access his credential. Anyone who requests Doctor Bob's AMA physician credential will be asked to disclose a physician credential, health-care organization credential, radiologist credential, or stem cell research credential. Bob's counter request might be enough evidence for a salesperson to target him for nuclear medicine products. Similarly, the requirement for a stem cell research credential might cause members in the Physicians for Life organization to assume Dr. Bob is a pro-choice physician. Thus, Dr. Bob may consider his policy as sensitive information and desire to limit its disclosure.

A trust negotiation can be simple and direct, as in the example involving Dr. Bob and JC. However, when both parties possess sensitive credentials or either party possesses a sensitive policy, a trust negotiation can expand to multiple rounds.

1.2 Motivation

Discussions of trust negotiation in the current research literature are often based on per-request access-control. Also, the current implementations of trust negotiation are designed to perform per-request access-control. A trust negotiation occurs every time a sensitive service or resource is requested. Even if the same resource or a resource with the same access control policy is requested multiple times, trust negotiation occurs for each request. This is obviously wasteful. If the resource requester has already authenticated sufficiently to receive a resource, why should he have to re-authenticate for the same resource or different resources with the same access control policy? In fact, why should he have to re-authenticate when all of his relevant credentials are still valid?

The lack of trust preservation means that if a client requests the same resource five times, the client and server negotiate trust five times. The client and server negotiate trust when a different resource is accessed, even if the resources have the same access control policy. Since trust is not preserved, this behavior applies to requests within a single session and across multiple sessions. In multi-session interactions, once a session is terminated, so is the relationship of trust between the participants. Similar to multi-session, trust is terminated after each request within a single session. Recent research proposes an approach to caching trust within a single session [8]. In contrast, this thesis focuses on preserving trust across multiple sessions.

The performance overhead for renegotiating trust increases when a negotiation involves multiple exchanges of policies and credentials. The cost of validating credentials increases linearly with the number of credentials received from the other negotiator. Network bandwidth and latency also increases linearly with the number of negotiation rounds. The overhead of renegotiating trust unnecessarily consumes network bandwidth and increases computational load. Preserving the trust already established between a client and server reduces these overheads.

The problem of trust preservation is similar to the single sign-on problem in identity-based authentication, where solutions are intended to avoid requiring users to enter a password repeatedly. The situation is different in trust negotiation because the goal is to fully automate the trust negotiation process. Automated trust negotiation takes place without direct user involvement. In this case, trust preservation is mostly a performance issue. However, there may be situations where fully automated trust negotiation is not feasible, and a user must be consulted to make authorization decisions, such as determining whether or not a credential is disclosed or determining which credentials to disclose. In this case, avoiding the need for user interaction during repeated negotiations is akin to the single sign-on problem for identity-based authentication systems.

1.3 Research goals

The goal of this research is to design and develop an approach for preserving trust in open systems that reduces the performance overhead to establish trust and eases the burden on users during semi-automated trust negotiation.

As proposed by [8], once a client and server negotiate trust, the server should retain the client's authentication information until the end of the session. Similarly, the client should cache the server's authentication information. By simply retaining relevant authentication information, the frequency of renegotiation can be vastly reduced.

Trust established during a client's first session with a server should be retained through subsequent sessions. It would also be convenient and more efficient for trust to persist across multiple sessions among servers in the same security domain or across trusted security domains.

Trust persistence promises to make authenticating strangers in open systems more efficient. Trust persistence also allows strangers to make friends—the relationship of trust persists, because the parties involved are no longer foreign to each other. Having authenticated each other once, they do not need to perform the same expensive level of authentication again.

1.4 Thesis statement

Trust is not persistent in the current trust negotiation model. Thus, a trusted principal is always required to reestablish trust on every subsequent request. This hinders the response time and throughput of a system in delivering resources to users.

In addition, this model decreases the battery lifetime of resource-constrained clients (e.g., PDAs and cell phones).

This thesis presents the design of two approaches for preserving trust across multiple sessions. Experimental prototypes of these two approaches were created using TrustBuilder, a prototype trust negotiation system under development in the Internet Security Research Lab at Brigham Young University (see http://isrl.cs.byu.edu). These prototypes illustrate two ways to deploy the current trust negotiation model.

The first approach, a stand-alone design that manages both authentication and authorization, introduces a Trust Resumption Token (TRT) that captures the details of a trust negotiation occurring in one session and makes it possible to reestablish trust during subsequent sessions more efficiently. The second approach is an integrated design to deploy TrustBuilder with an existing access control system. Many authentication and authorization systems are identity-based. In contrast to TrustBuilder, these identity-based systems are not designed to authenticate or authorize strangers. This integrated approach enables these existing systems to authenticate and authorize strangers.

The focus of this thesis is to develop approaches to preserving trust. The integration of TrustBuilder with existing systems eases the deployment of trust negotiation technology. Trust preservation reduces the cost of credential validation during subsequent visits from a cost that increases linearly according to the number of credentials received to a cost that is constant regardless of the number of credentials involved in the negotiation. It also consolidates multiple rounds of negotiation into an exchange of one policy and one token. Trust preservation increases the throughput of systems delivering resources to users, resulting in a decrease in response time to users.

1.5 Outline

The remainder of this thesis is organized as follows: Chapter 2 presents the design and implementation of the trust resumption token. Chapter 3 describes the design of the integration of TrustBuilder with Kerberos, a popular authentication system. It also presents the design and implementation of the integration of TrustBuilder with KeyNote, an existing authorization system. Chapter 4 contains a performance analysis of the TRT implementation. Related work is presented in Chapter 5. The final chapter contains conclusions and proposes future research directions.

Chapter 2 — Trust Preservation using a TRT

An approach to preserving trust across multiple sessions is to use a certified token known as an authenticator. A server can issue an authenticator to a client following a successful negotiation. The client can use the token to efficiently reestablish trust with the server during subsequent sessions. This thesis refers to such an authenticator as a Trust Resumption Token (TRT).

This chapter presents some design goals for a TRT and discusses several design alternatives for using a trust resumption token to preserve the trust that is established between two negotiators across multiple sessions.

Figure 2.1 illustrates an extension of the previous medical example of a trust negotiation between Dr. Bob and JC. Suppose Alice's condition intensifies the following day, and Dr. Bob needs to review a different medical record of Alice before he prescribes different medicine and treatment to her. With the new design, Bob acquires an authenticator from JC's server along with Alice's records the first time he has successfully negotiated with JC. Now, he can re-access Alice's records using the authenticator and eliminate the need to re-negotiate with JC's server.

2.1 Design goals

The following are the design goals for using a TRT:

- Integrity: detect when data in the TRT has been altered
- Authenticity: obtain evidence that the client submitting the TRT is its rightful owner
- Validation: validate that the corresponding credentials that were used to issue the token are still valid and have not been revoked

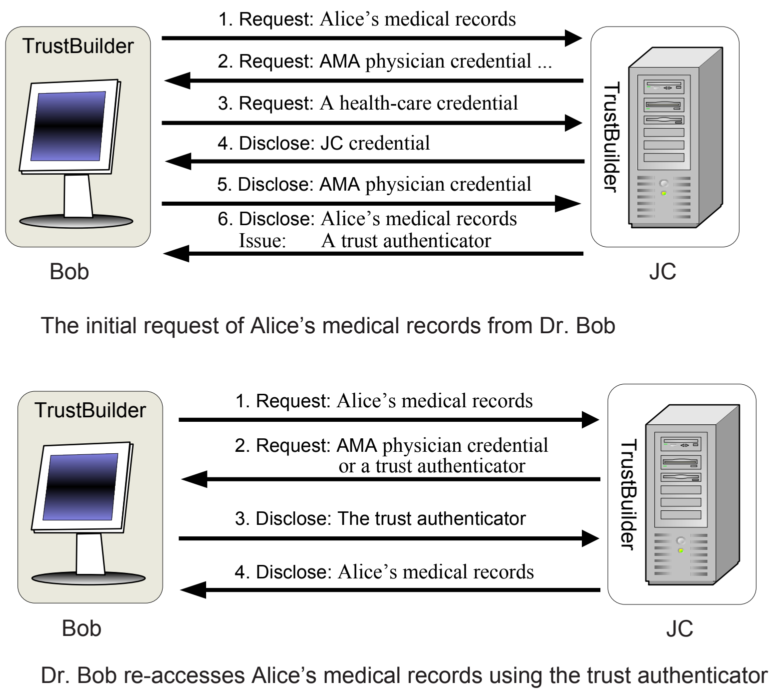

TrustBuilder
1. Request: Alice's medical records
2. Request: AMA physician credential ...
3. Request: A health-care credential
4. Disclose: JC credential
5. Disclose: AMA physician credential
6. Disclose: Alice's medical records
Issue: A trust authenticator
TrustBuilder
Bob
JC
The initial request of Alice's medical records from Dr. Bob
TrustBuilder
1. Request: Alice's medical records
2. Request: AMA physician credential or a trust authenticator
3. Disclose: The trust authenticator
4. Disclose: Alice's medical records
TrustBuilder
Bob
JC
Dr. Bob re-accesses Alice's medical records using the trust authenticator

Figure 2.1: An extended example to illustrate the usage of an authenticator

- Privacy or confidentiality: limit the disclosure of sensitive attributes in the token to the server

- Performance: reduce the computational requirements for trust negotiation

A server can digitally sign the TRT when it is issued so that the integrity of a TRT can be determined when a client presents it to a server in the future. A TRT can be implemented using X.509v3 certificates.

A server could include the following attributes in the TRT: relevant attributes of the user, a list of unique public keys, a list of serial numbers and issuer names, and an expiration date. The following discussion addresses the benefits of storing attributes of the user, and how the other items contribute to the integrity, authenticity, confidentiality, and validity of the TRT.

First, the attributes that were initially required to access the requested resource could be included in the TRT. Servers that receive the TRT in the future can use these attributes to determine what resources the requester is entitled to access. This approach enables the access privileges associated with the token to dynamically evolve as an access control policy changes. Compared to storing permissions in a TRT, this reduces the threat of requesters retaining the privileges associated with the original policy when a policy update occurs. It also lessens the need to revoke tokens when policies are modified.

Sometimes, a TRT can contain sensitive attributes or roles. In such cases, the privacy and confidentiality of this sensitive information can be protected in different ways. For example, the server could encrypt this information using the server's public key or a symmetric key known only by the server. This would prevent unauthorized access by an eavesdropper or an attacker that gains access to the TRT stored on disk. Also, a server that does not need to maintain a permanent copy of a client credential could avoid storing it on disk to reduce the server's liability and to reduce the risk of the server becoming a prime target of an identity thief.

A server accepting a TRT must determine whether the client submitting the TRT is the legitimate owner. One approach to determining ownership of the TRT is to link the TRT to the original certificates that were instrumental to the issuing of the TRT. The public keys contained in the original certificates are stored in the TRT. When the TRT is presented, the server can challenge the client for the corresponding private keys. Another approach is to generate a new public/private key pair specific to the TRT and store the public key in the TRT. The server can challenge for ownership of the private key when the TRT is presented.

The certificates used to authenticate the client in order for the TRT to be issued could be revoked because one of the attribute values has changed or become invalid.

A server may not want to accept the TRT when this occurs. One approach to detect this situation is to include a list of serial numbers and issuer names in the TRT from the relevant credentials that caused the TRT to be issued. The server can verify if any of the asserted attributes are still valid by acquiring the certificate revocation lists (CRLs) that correspond to the source credentials.

The certificates used to authenticate the client expire at various times. The expiration date of the TRT can be set to be the expiration date of the source credential that expires first. This ensures that the TRT is only valid so long as all of the source credentials have not expired.

A difficult issue in certificate systems is the lending problem. This refers to the unauthorized sharing of security tokens. The general problem is beyond the scope of this thesis. However, in some cases challenging the ownership of the tokens can discourage owners from lending tokens to other parties especially when sensitive attributes are stored in these tokens. The ownership challenge process requires the bearer to encrypt an unpredictable message with private keys that are associated with the public keys included in the token. As users would also need to share private keys to accomplish the challenge, they may be less inclined to share their tokens.

The information contained in a trust negotiation authenticator could be based on attributes, roles, or permissions. Attributes are characteristics of an individual. They are commonly contained in a digital credential certified by an authority (a trusted third party). Roles typically represent various functions in an organization and are granted certain permissions [14]. Permissions are the ability to access a resource or a service. In trust negotiation, attributes are mapped to roles, and each role is associated with a set of permissions. The design decision of what information to store in an authenticator must consider tradeoffs between computation cost, storage requirements, and the frequency of renegotiation.

For example, suppose that service S_1 can be accessed by role R_1, and service S_2 can be accessed by role R_2. Attribute A_1 plus attribute A_2 or attribute A_3 alone can authenticate to R_1, whereas A_1 alone can authenticate to R_2. Assume client C succeeds in requesting access for S_1, and receives an authenticator for future access.

If the authenticator contains only the roles to which a client has authenticated, when the client attempts to access a service requiring a different role, the server may be unable to determine whether the client has already presented enough information to authenticate to the role, even if it requires some of the same attributes. Suppose client C tries to access S_2. He cannot re-use the authenticator since the server does not know whether the authenticator was issued based on attribute A_1 or not. A role-based authenticator works well when future client requests require authentication to the same role. In a more dynamic environment where a client assumes various roles based on different attributes, role-based authenticators may not be the most efficient option.

If the authenticator contains only the attributes the client used to authenticate to a role, when the client attempts to access a different service the server can simply determine which roles are required to access the service and determine whether the stored attributes permit or qualify the client to authenticate to those roles. Suppose client C tries to access S_2. The only required attribute to authenticate to role R_2 is A_1. Thus, client C can access R_2 only when the authenticator contains A_1. Unlike role-based authenticators, attribute-based authenticators always require the additional mapping between attributes and roles. An attribute-based authenticator preserves the authenticated attributes from a client—thus preserving the possibility of mapping those attributes to other roles during future requests.

2.2 Trust resumption token

In this thesis, the design for preserving trust calls for attributes to be stored in an authenticator rather than storing roles or permissions. This thesis further proposes the use of signed digital credentials as the logical structure for trust authenticators.

During a single trust negotiation, the server extracts attributes and other information from the credentials disclosed by the client. If the negotiation succeeds, the server generates and signs a credential—the *trust resumption token*—and delivers it to the client along with the requested resource. A trust resumption token contains the following: those attributes that were required to access a secure resource, a list of unique public keys that are associated with the attributes, a list of serial numbers and issuer names from the relevant credentials, and the expiration date from the credential that expires first. With this token, a client can re-access the server and quickly reestablish the same degree of trust that was previously established, avoiding needless repetition of a lengthy negotiation process.

Trust resumption tokens contain attributes rather than roles or permissions for the following reasons. In trust negotiation, attributes are the lowest common denominator of trust. The mapping from attributes to roles obviously requires attributes, and the mapping from roles to permissions is therefore also based on attributes. Because attributes are the basis of trust, attributes are less likely to change than roles or permissions. The definition of a role contains a logic expression based on attributes; that logic expression may change according to the security needs of the system. The issue of changes to a role definition also applies to permissions. If a TRT contains permission to access a particular file on the server and then the access-control policy for the file changes, the holder of the TRT can still access the file until either the TRT is revoked or it expires, regardless of the updated access-control policy. Changes to the definition of a role or permission introduce additional insecurities into the system.

If roles or permissions are stored in a TRT, revocation of the TRT becomes a serious issue. When there is a change in a role definition or the permissions to access a resource, any holder of a TRT issued previously would still be able to access the resource even though they may no longer be authorized to hold that role or exercise the permission. One way to mitigate this risk is to limit the lifetime of a TRT.

2.2.1 TRT example

The following example illustrates how a TRT is used to resume trust. Suppose Alice requests service S from Bob, and the access control policy for S requires that Alice possess attributes A_1, A_2, and A_3.

Alice's C_1 credential contains A_1, and A_2. A_3 is contained in Alice's C_2 credential. According to Alice's access control policy, C_1 is freely available only upon request and C_2 is only to be disclosed to a requestor who can demonstrate ownership of attribute A_4. Alice discloses her C_1 credential to Bob and requests proof that Bob has attribute A_4. Bob indeed possesses A_4 in his C_3 credential, and it is freely available only upon request. Bob collects Alice's C_1 and discloses his C_3 to Alice. Alice discloses her C_2 credential after she receives Bob's C_3 credential. Upon receiving Alice's C_2 credential, Bob generates a TRT that contains attributes A_1, A_2, and A_3, both issuer names and public keys of the credentials, and the expiration date of the credential which will expire the soonest. Then Bob grants Alice access to S and sends her the TRT. Figure 2.2 shows the flow of the negotiation between Alice and Bob.

Trust negotiation can involve additional credentials and more rounds of negotiation than this simple example illustrates. Using the TRT during a later session, a client can resume the same level of trust with only one credential and one round of negotiation. Figure 2.3 shows the flow of a negotiation between Alice and Bob with more rounds of negotiation.

Based on the given Alice and Bob example, the TRT allows Alice to re-access

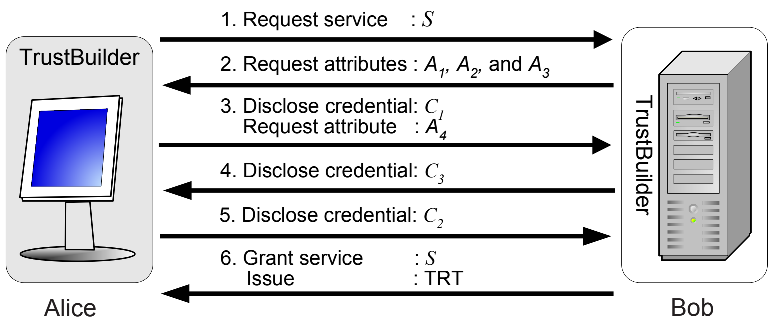

TrustBuilder
Alice
1. Request service : S
2. Request attributes : A_1, A_2, and A_3
3. Disclose credential: C_1
Request attribute : A_4
4. Disclose credential: C_3
5. Disclose credential: C_2
6. Grant service : S
Issue : TRT
TrustBuilder
Bob

Figure 2.2: The trust negotiation flow between Alice and Bob.

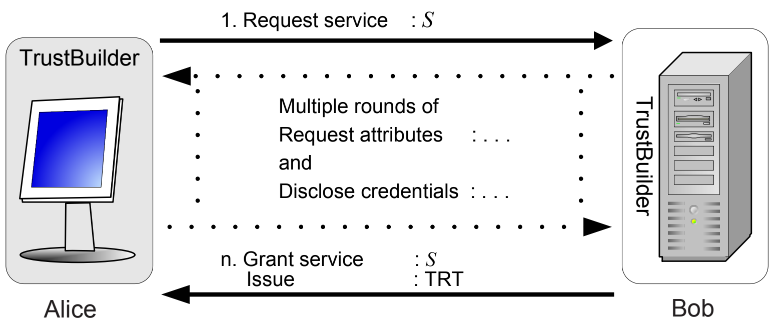
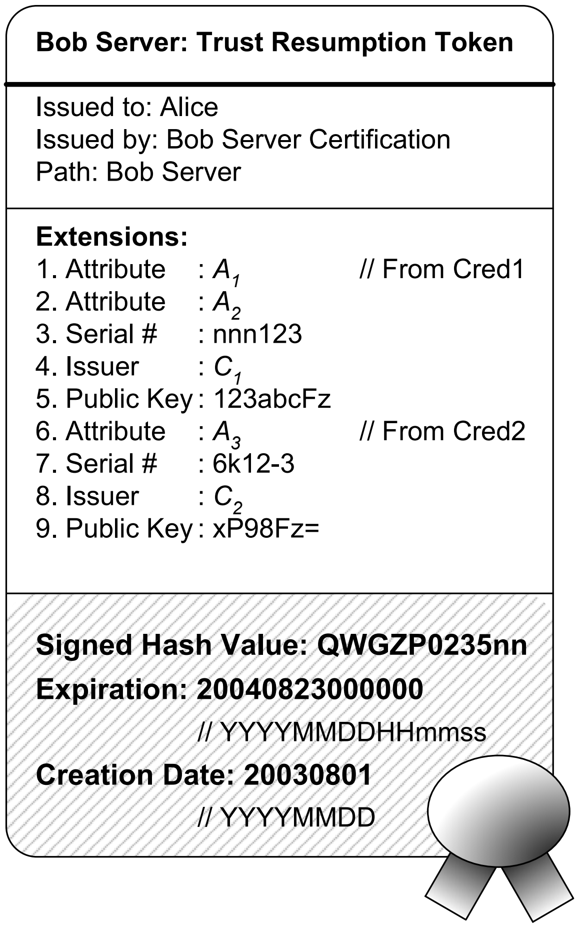


Figure 2.3: The trust negotiation flow between Alice and Bob with multiple rounds of negotiation.



Bob Server: Trust Resumption Token

Issued to: Alice
Issued by: Bob Server Certification
Path: Bob Server

Extensions:

1. Attribute	: A_1	// From Cred1
2. Attribute	: A_2	
3. Serial #	: nnn123	
4. Issuer	: C_1	
5. Public Key	: 123abcFz	
6. Attribute	: A_3	// From Cred2
7. Serial #	: 6k12-3	
8. Issuer	: C_2	
9. Public Key	: xP98Fz=	

Signed Hash Value: QWGZP0235nn
Expiration: 20040823000000
// YYYYMMDDHHmmss
Creation Date: 20030801
// YYYYMMDD

Figure 2.4: Alice's TRT

service S and all other services with the same access control policy, or a subset of the policy. The TRT illustrated in Figure 2.4 is a simplified version for demonstration purposes only. It does not include all the necessary fields in a typical X.509v3 certificate.

Figure 2.5 shows Alice using the TRT to resume the level of trust she established previously with Bob. In this figure, Alice requests the same service S from Bob. Bob replies to Alice with the access control policy for the service. As before, Bob's access control policy requires attributes A_1. A_2, and A_3. Instead of sending credentials C_1

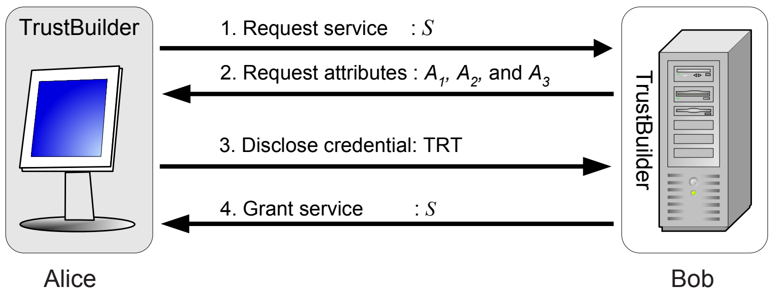


Figure 2.5: Alice resumes the trust level using TRT.

and the access control policies for credential C_2 to Bob, Alice replies with the TRT that Bob previously issued to her. Upon receiving Alice's TRT, Bob checks that the TRT is valid, and then checks it against the access control policy before granting Alice access to S.

Bob checks the validity of the TRT by verifying its signature, verifying that it has not expired, and verifying that Alice owns it by challenging Alice for possession of the private keys associated with the public keys contained in the TRT. Bob can also verify that the source credentials used to obtain the attributes in the TRT have not been revoked by checking the certificate revocation lists for the credentials using the Issuer and Serial # fields contained in the token.

2.2.2 TRT usage in a federation

The potential for multiple servers to join in an alliance to share authentication information introduces another opportunity to utilize a TRT. Alliance servers have a preexisting trust relationship with the token's issuer, similar to many single sign-on paradigms like the Liberty Alliance Project [7] and Microsoft Passport [11]. In contrast to the Liberty Alliance Project and Microsoft Passport, trust-negotiation-enabled single sign-on is attribute-based.

For example, if a group of trusted servers adopt a common set of roles, then a server could issue a TRT to designate that the client satisfies a particular role, and other servers could accept the TRT for rapid authentication to the same role in their local domain. Another approach is for the servers to accept a common set of attributes and map clients to local roles based on those attributes. An approach that combines both attributes and roles in the TRT is also possible.

Chapter 3 — Trust Preservation using Kerberos and KeyNote

A fundamental task for a system that provides protected services and resources is to determine the trustworthiness of clients that access the system. Traditional approaches to establishing trust assume that the interacting parties are already familiar with each other. Under this assumption, there are essentially two approaches. The first approach is the identity-based approach, which identifies whether a client is indeed who it claims to be. Kerberos is an example of a system based on this approach. The second approach is the capability-based approach, which determines if a client has the capability (authority) to access the services or resources it is requesting. KeyNote is an example of a system based on this approach.

Traditional approaches to trust establishment are not suitable for establishing trust between complete strangers. This chapter describes how existing systems can leverage trust negotiation to establish trust with complete strangers, and how trust negotiation systems, like TrustBuilder, can leverage existing systems to preserve trust.

3.1 Kerberos integration

Kerberos is a distributed identity-based authentication system developed at MIT. Figure 3.1 illustrates the system architecture. This section presents a high-level design of integrating Kerberos with TrustBuilder for preserving trust across multiple sessions and describes how this integration can open Kerberos services to strangers with qualified attributes. This integration of TrustBuilder with Kerberos allows strangers to obtain a limited form of trust preservation, which makes it possible for these currently authenticated strangers to reestablish trust during subsequent sessions and to request other services within a realm more efficiently.

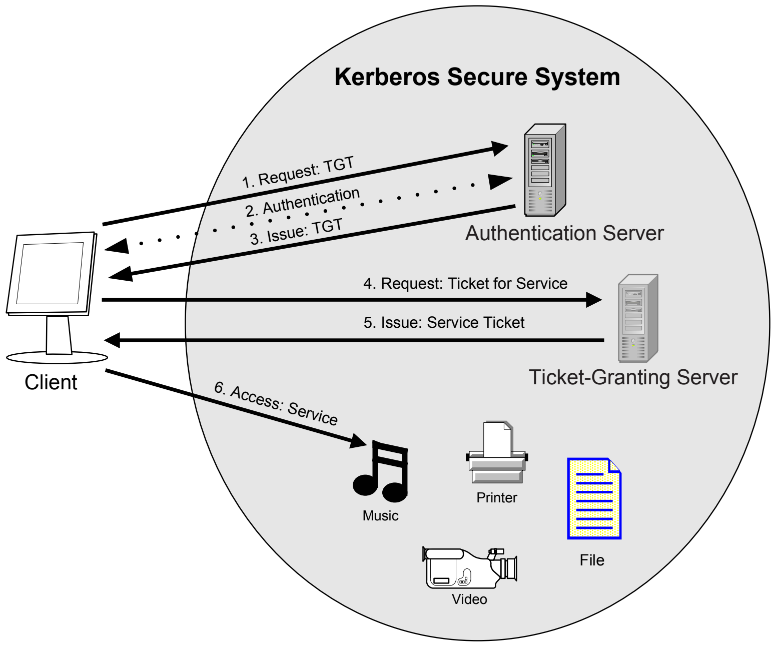


Figure 3.1: The Kerberos distributed authentication system architecture.

Kerberos was originally designed for closed systems; the client must have a pre-existing relationship with the Kerberos Authentication Server. Typically, Kerberos users need to establish and obtain their Kerberos username and password prior to accessing any services that are protected behind the Kerberos system. In Kerberos, a ticket is required for a user to access a service. Kerberos manages tickets with a key distribution center (KDC), which consists of an authentication server (AS) and a ticket-granting server (TGS). After a client is authenticated to the AS with its Kerberos username and password, the AS issues a ticket-granting ticket (TGT) to the client. Later, the client can use the TGT to request access to different services through the TGS. Authentication to the AS is preserved within the TGT. As such, an authenticated client does not have to re-authenticate to the AS in order to access other services. Instead, clients can request other services by requesting a service granting ticket from the TGS, as long as their TGT is still valid and has not expired.

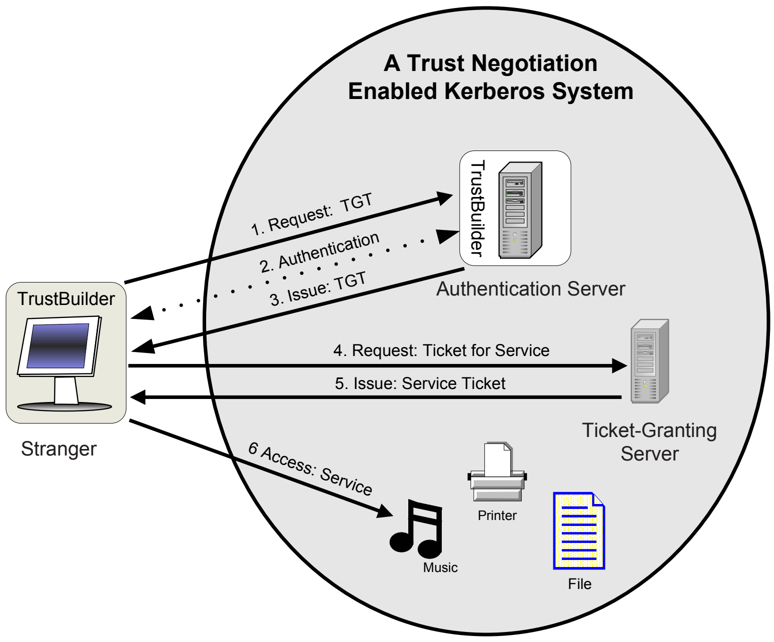


Figure 3.2: A simple design for integrating TrustBuilder with Kerberos.

This two-level design eliminates the need for authenticated clients to reenter their Kerberos password for every requested service [9].

The process of establishing and obtaining a Kerberos user name and password is typically based on identity. As it has been stated previously, this process is not practical in an open system where users have no pre-existing relationship. Kerberos can be extended to support authentication of strangers by integrating trust negotiation into the initial authentication step between a Kerberos client and the AS, as shown in Figure 3.2.

One approach to integrating Kerberos and TrustBuilder is to embed TrustBuilder in the AS module. Hereafter, we will refer to the AS module extended with Trust-Builder as a trust-negotiation-enabled AS (TNAS). This enhanced AS permits a stranger to negotiate a service through a negotiation process similar to what was described in Chapter 1. Upon a successful negotiation, TNAS maps a user's submitted attributes to a pseudo-id that corresponds to the requested role of the user. The TNAS returns a TGT for the pseudo-id that entitles the user to access services in the Kerberos realm for which the pseudo-id is authorized. This approach opens up Kerberos services to users outside the Kerberos realm.

In this design, TrustBuilder is used to interface with all the strangers outside the Kerberos realm. The TGT obtained through TrustBuilder preserves the trust that is established during the initial trust negotiation between the user and the TNAS. Since the TGT is reusable until it expires, the authenticated strangers can use the TGT to request other services within the Kerberos realm that the pseudo-id is authorized to access.

3.2 KeyNote integration

KeyNote is a capability-based trust management system designed to assist applications in authorizing user's requests. Figure 3.3 illustrates how KeyNote can be used to answer the following question: should the requested action by this subject be permitted (or authorized) for a given policy and a set of credentials? The task of authenticating the subject is left to the application.

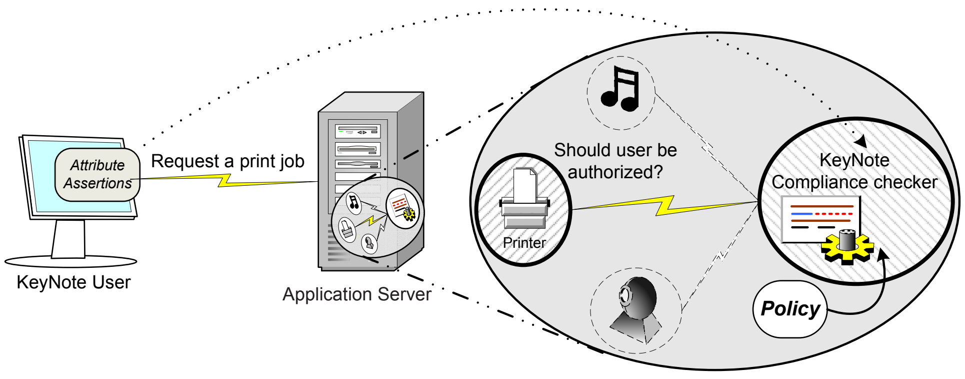
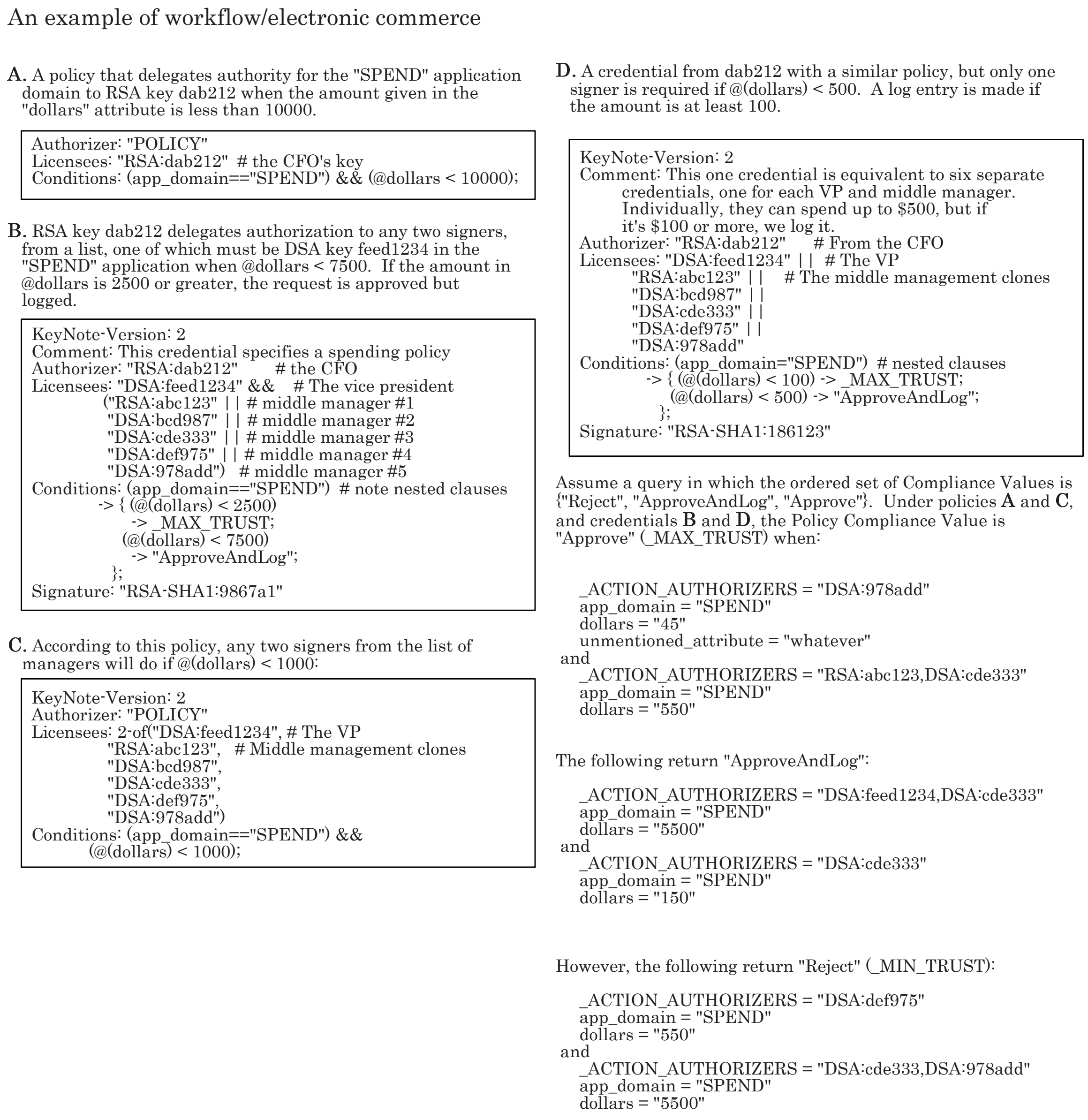


Figure 3.3: Applications use KeyNote to determine whether a client is authorized for a requested service.

KeyNote is a general-purpose, distributed trust management system that is designed for access control policies to be distributed over a network or otherwise decentralized [1–3]. It uses a unified format and a highly expressive language to describe its access control policies and credentials. Figure 3.4 is an example taken from [1], that illustrates how KeyNote access control policies and credentials can be used in a spending application. In this example, A is the root policy and C is an extended policy. These policies typically remain local to the application. B and C are signed policies; they are often referred to as credential assertions or simply credentials. These credentials are issued and signed by a trusted principal, or in this case, the Chief Financial Officer (CFO). According to the root policy, the CFO is a trusted principal that is authorized to make expenditures under \$10,000. As shown in Figure 3.4, KeyNote credentials and policies share the same structure and syntax. KeyNote credentials provide a vehicle for a trusted principal to delegate trust and authority to other principals. The credentials are signed to ensure integrity so that they can be distributed over an untrusted network.

KeyNote policies and credentials are collectively known as assertions. A KeyNote credential is capability-based, i.e., it describes authorized actions, and can provide specific delegation of trust to other principals. A credential also identifies the delegator. KeyNote assertions are necessary for a user to access a secure service. In order for an application to make use of KeyNote, it needs to provide the user's identity (_ACTION_AUTHORIZERS), the application's identifier (app_domain), action description, a set of policies for the application, and a set of credential assertions for KeyNote to determine if the user is in compliance with the access control policies. The example in Figure 3.5 depicts the process of a manager, with a public key RSA:abc123, requesting a \$99 expenditure from a spending application, and the authorization query made by the spending application to KeyNote.



An example of workflow/electronic commerce

A. A policy that delegates authority for the "SPEND" application domain to RSA key dab212 when the amount given in the "dollars" attribute is less than 10000.

```
Authorizer: "POLICY"
Licensees: "RSA:dab212"  # the CFO's key
Conditions: (app_domain=="SPEND") && (@dollars < 10000);
```

B. RSA key dab212 delegates authorization to any two signers, from a list, one of which must be DSA key feed1234 in the "SPEND" application when @dollars < 7500. If the amount in @dollars is 2500 or greater, the request is approved but logged.

```
KeyNote-Version: 2
Comment: This credential specifies a spending policy
Authorizer: "RSA:dab212"        # the CFO
Licensees: "DSA:feed1234" &&    # The vice president
             ("RSA:abc123" | | # middle manager #1
              "DSA:bcd987" | | # middle manager #2
              "DSA:cde333" | | # middle manager #3
              "DSA:def975" | | # middle manager #4
              "DSA:978add")   # middle manager #5
Conditions: (app_domain=="SPEND")  # note nested clauses
            -> { (@(dollars) < 2500)
                  -> _MAX_TRUST;
                (@(dollars) < 7500)
                  -> "ApproveAndLog";
               };
Signature: "RSA-SHA1:9867a1"
```

C. According to this policy, any two signers from the list of managers will do if @(dollars) < 1000:

```
KeyNote-Version: 2
Authorizer: "POLICY"
Licensees: 2-of("DSA:feed1234", # The VP
              "RSA:abc123",   # Middle management clones
              "DSA:bcd987",
              "DSA:cde333",
              "DSA:def975",
              "DSA:978add")
Conditions: (app_domain=="SPEND") &&
          (@(dollars) < 1000);
```

D. A credential from dab212 with a similar policy, but only one signer is required if @(dollars) < 500. A log entry is made if the amount is at least 100.

```
KeyNote-Version: 2
Comment: This one credential is equivalent to six separate
        credentials, one for each VP and middle manager.
        Individually, they can spend up to $500, but if
        it's $100 or more, we log it.
Authorizer: "RSA:dab212"      # From the CFO
Licensees: "DSA:feed1234" | |  # The VP
          "RSA:abc123" | |     # The middle management clones
          "DSA:bcd987" | |
          "DSA:cde333" | |
          "DSA:def975" | |
          "DSA:978add"
Conditions: (app_domain="SPEND")  # nested clauses
            -> { (@(dollars) < 100) -> _MAX_TRUST;
                 (@(dollars) < 500) -> "ApproveAndLog";
               };
Signature: "RSA-SHA1:186123"
```

Assume a query in which the ordered set of Compliance Values is {"Reject", "ApproveAndLog", "Approve"}. Under policies **A** and **C**, and credentials **B** and **D**, the Policy Compliance Value is "Approve" (_MAX_TRUST) when:

```
    _ACTION_AUTHORIZERS = "DSA:978add"
    app_domain = "SPEND"
    dollars = "45"
    unmentioned_attribute = "whatever"
 and
    _ACTION_AUTHORIZERS = "RSA:abc123,DSA:cde333"
    app_domain = "SPEND"
    dollars = "550"
```

The following return "ApproveAndLog":

```
    _ACTION_AUTHORIZERS = "DSA:feed1234,DSA:cde333"
    app_domain = "SPEND"
    dollars = "5500"
 and
    _ACTION_AUTHORIZERS = "DSA:cde333"
    app_domain = "SPEND"
    dollars = "150"
```

However, the following return "Reject" (_MIN_TRUST):

```
    _ACTION_AUTHORIZERS = "DSA:def975"
    app_domain = "SPEND"
    dollars = "550"
 and
    _ACTION_AUTHORIZERS = "DSA:cde333,DSA:978add"
    app_domain = "SPEND"
    dollars = "5500"
```

Figure 3.4: A spending application using KeyNote to determine a user's spending power.

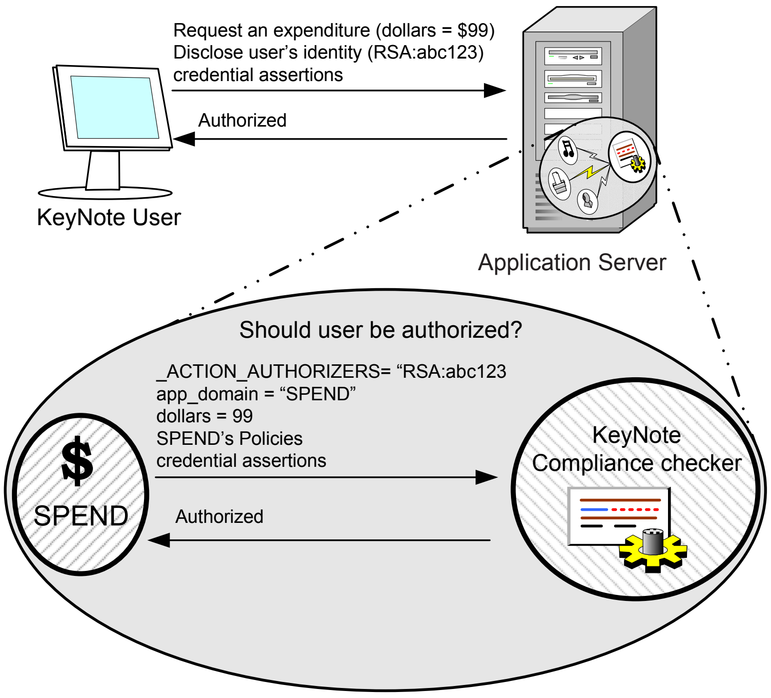


Figure 3.5: A typical flow of interaction that illustrates an application making a query to KeyNote based on a user's request.

The following design is one way to integrate TrustBuilder with KeyNote. In this design, TrustBuilder will act as proxy for each application to negotiate with strangers given an attribute-based policy which contains the required attributes for accessing each application. For example, suppose a jukebox application is serving music on demand to any full-time student. David is a fulltime student that is a stranger to the jukebox application. David negotiates with TrustBuilder for access to the music server. Once David's negotiation is successful, TrustBuilder delegates the capability for accessing the service to David by issuing him a KeyNote credential assertion. This credential assertion would look like the following:

```
KeyNote-Version: 2

Comment: This credential is issued to a stranger who is authenticated
     by TrustBuilder, and approved to access the requested music

Authorizer: "RSA:TB-Key"          \# From TrustBuilder

Licensees: "DSA:David-Key"        \# The authenticated stranger

Conditions: (app_domain="music")  \# requested song from the music server

Signature: "RSA-SHA1:186123"      \# signed by TrustBuilder
```

By integrating TrustBuilder with KeyNote, strangers with qualified attributes can be authenticated through trust negotiation. Then, TrustBuilder can issue a KeyNote credential assertion to the authenticated user. With this credential assertion, a user is authorized to access resources without repeating the negotiation process. The integration of trust negotiation with KeyNote is the first working example of trust negotiation integrated with an existing authorization system.

The implementation of trust preservation for this section is based on KeyNote version 2 and TrustBuilder version 1. Binary and source files for KeyNote are freely

```
⋮ ⋮ ⋮
int    kn_init(void);
int    kn_add_assertion(int, char *, int, int);
int    kn_remove_assertion(int, int);
int    kn_add_action(int, char *, char *, int);
int    kn_remove_action(int, char *);
int    kn_add_authorizer(int, char *);
int    kn_remove_authorizer(int, char *);
int    kn_do_query(int, char **, int);
int    kn_get_failed(int, int, int);
int    kn_cleanup_action_environment(int);
int    kn_close(int);
void   kn_free_key(struct keynote_deckey *);
char  *kn_get_string(char *);

/* Simple API */
int    kn_query(struct environment *, char **, int, char **, int *, int, char **, int *, int, char **, int);

/* Aux. routines */
char **kn_read_asserts(char *, int, int *);
int    kn_keycompare(void *, void *, int);
void  *kn_get_authorizer(int, int, int *);
struct keynote_keylist *kn_get_licensees(int, int);

/* ASCII-encoding API */
int    kn_encode_base64(unsigned char const *, unsigned int, char *, unsigned int);
int    kn_decode_base64(char const *, unsigned char *, unsigned int);
int    kn_encode_hex(unsigned char *, char **, int);
int    kn_decode_hex(char *, char **);

/* Key-encoding API */
int    kn_decode_key(struct keynote_deckey *, char *, int);
char  *kn_encode_key(struct keynote_deckey *, int, int, int);

/* Crypto API */
char  *kn_sign_assertion(char *, int, char *, char *, int);
int    kn_verify_assertion(char *, int);
#endif /* __KEYNOTE_H__ */
```

Figure 3.6: The KeyNote API.

available from the KeyNote Web Page at http://www.cis.upenn.edu/˜keynote/. At the time of this writing, all the available versions and implementations of KeyNote are written in the C programming language. In contrast to KeyNote, TrustBuilder is written in Java. In order for TrustBuilder to interact with KeyNote, a KeyNote-wrapper class was written using the Java Native Interface (JNI) based on the KeyNote Application Programming Interface (API). The KeyNote API can be located in Keynote.h, one of the header files. Figure 3.6 shows a subset of the API that has been implemented as the KeyNote-wrapper class. This KeyNote-wrapper class allows TrustBuilder to interface with KeyNote programmatically. This API provides a way for TrustBuilder to generate a credential assertion for a user. The API also permits TrustBuilder to query KeyNote to determine whether a user is compliant with the given policy provided by the application.

Current literature does not address how a particular application user can automatically acquire a KeyNote assertion for accessing services provided by the application. The integration of TrustBuilder with KeyNote allows the issuing of assertions to strangers on demand. This chapter discussed two different approaches to integrate existing systems with TrustBuilder. This chapter also covered the design of integrating TrustBuilder with Kerberos and KeyNote, showed how a complete stranger can authenticate to a Kerberos system to acquire a Kerberos ticket, and how a stranger can acquire a KeyNote assertion on demand.

Chapter 4 — Performance Analysis

This chapter contains a performance analysis of an extension to TrustBuilder that supports the trust resumption token described in Chapter 2. This extension enables TrustBuilder to preserve trust once it is established via trust negotiation. The extended version of TrustBuilder is referred to as TB-TRT in the remainder of this chapter. TB-TRT includes the following extensions to TrustBuilder:

- Server issues a resumption token to clients after a successful negotiation
- Client stores the resumption token on local machine
- Client uses corresponding resumption token to request a resource from a server
- Server allows a client to access resources with a valid resumption token

The performance analysis in this chapter compares TB-TRT to the use of TrustBuilder without a TRT, known as TB-noTRT. The results will demonstrate circumstances where a server can increase its throughput using a TRT, resulting in a reduction in the response time experienced by clients.

The following sections in this chapter are organized as follows. Section 4.1 describes the characteristics of the test platform and the test suites used to measure the throughput and response time of the TB-TRT and TB-noTRT systems. Section 4.2 presents the performance analysis results of the system throughput and response time of the TB-TRT and TB-noTRT systems.

4.1 Experimental framework

The performance analysis was conducted using a typical client/server architecture. The client and the server were Intel-based Pentium 4 (P4) machines connected directly

	Client	Server
CPU	P4 1.4GHz	P4 1.7GHz
Memory	512 MB	512 MB
Network	100-Mbps	100-Mbps

Table 4.1: Client and server specifications

to a 100-Mbps Ethernet segment. The characteristics of these test platforms in terms of their computation power, physical memory and network speed are summarized in Table 4.1.

JMeter is a load and performance testing application available through the Apache Jakarta Project (see http://jakarta.apache.org/jmeter/index.html). JMeter uses a full multi-threading framework to simulate multi-user requests. JMeter was installed and run on the client to assist in analyzing the performance of TrustBuilder under various load conditions.

Using JMeter, five test groups were defined consisting of a group of users simultaneously requesting a protected resource from the server. The number of users in a group range from five to twenty-five, in increments of five. Each user in a test group repeats the same request to the server one hundred times with a one second delay between each request. Figure 4.1 depicts how these test groups interact with the server.

Three variations of TrustBuilder were evaluated on the hardware configuration described earlier for each of the five test groups. The first variation of TrustBuilder was TB-noTRT, which does not support any trust preservation. Users are always treated as strangers, and a trust negotiation process is repeated each time a user requests a protected resource. This occurs even when the same user is repeatedly

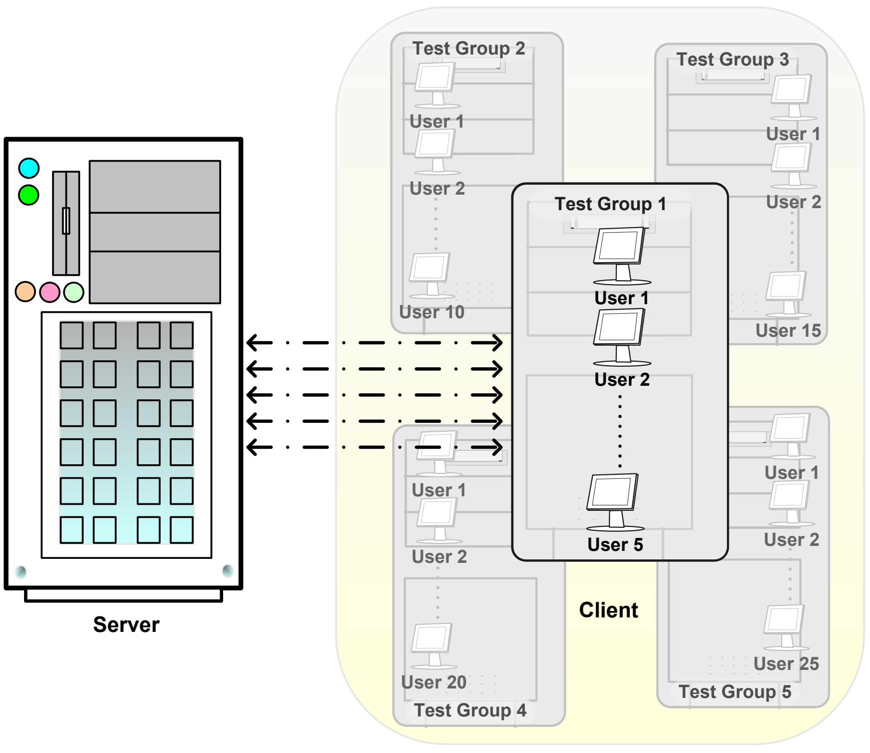


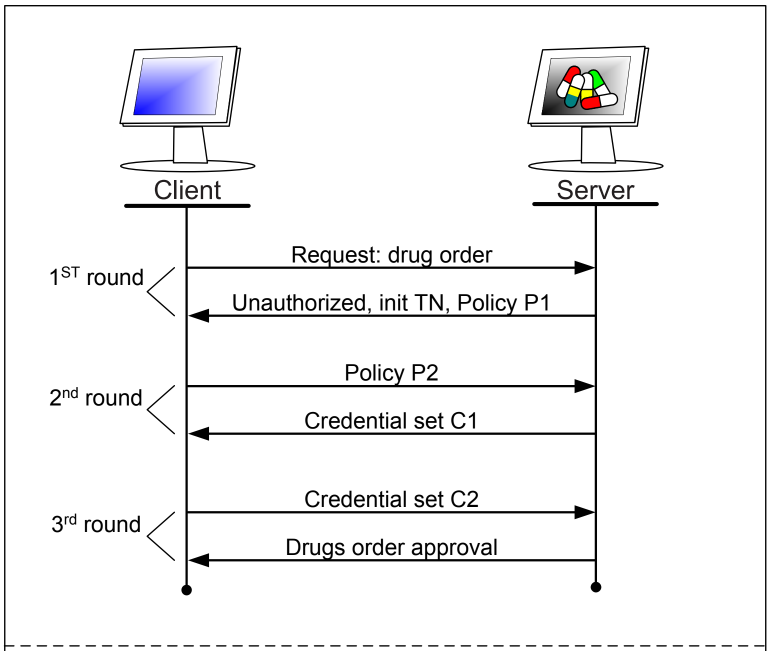
Figure 4.1: Various sized groups of users simultaneously negotiating trust for a protected resource.

requesting the same resource.

The second variation of TrustBuilder was TB-TRT, with fully functional TRT processing running at the server to issue trust resumption tokens. The client machine was equipped with a semi-functional TRT capability that accepts a TRT issued from the server and stores it at the client. However, the client does not submit a TRT on subsequent visits to the server. The purpose for this variation is to measure the overhead of generating the trust resumption tokens. Hereafter, this system is referred to as TB-TRT-issue.

The third variation of TrustBuilder was also TB-TRT, with fully functional TRT processing at both the client and server. A resumption token is pre-installed on the client to permit the client to submit the TRT in every request to the server. The purpose for this variation is to measure the throughput and response time when clients return to a server and submit a TRT. Hereafter, this system is referred to as TB-TRT-reuse.

All the experiments utilized a pharmacy trust negotiation scenario that involved a three-round trust negotiation. The pharmacy scenario is illustrated in Figure 4.2. In this scenario, the pharmacy (client) places a drug order from a pharmaceutical company (server). The drug order is a sensitive resource, which the pharmaceutical company only releases to parties with a valid pharmacy credential. The client possesses a valid pharmacy credential but will not disclose it without receiving a valid pharmaceutical credential from the server. The server is willing to freely release its pharmaceutical credential as proof to the pharmacy. Once received, the credential is validated, which allows the client to release its sensitive credential. The server verifies the received pharmacy credential which then fulfills the policy for the initial drug order.



Note:
TN: Trust negotiation
P1 : Drug order requires a valid pharmacy credential
P2 : Valid pharmacy credentials requires a valid Pharmaceuticals Company credential
C1 : Pharmaceuticals Company credential
C2 : Pharmacy credential

Figure 4.2: A pharmacy scenario involving a three-round trust negotiation process.

4.2 Results: throughput and response time

The pharmacy scenario described above was repeated one hundred times by each user in each of the five test groups, for the three variations of TrustBuilder (TB-noTRT, TB-TRT-issue, TB-TRT-reuse).

The throughput of the server for each group was recorded. Throughput is defined as the number of requests fulfilled by the server per minute. Based on the collected data, the maximum throughput for a server running TB-noTRT is 107 requests per minute. The maximum throughput for a server running TB-TRT-issue is 99 requests per minute. The maximum throughput for a server running TB-TRT-reuse is approximately 180 requests per minute. These results are shown in Figure 4.3.

The results indicate that the overhead of generating a resumption token to clients upon a successful negotiation is modest compared to the increased throughput when using the trust resumption token to resume trust during a later session with the same server. This has the potential for a significant performance improvement at the server when clients make repeat visits.

The mean execution (response) time for each client is the time from when a request for a protected resource is issued by a client to the time that the requested resource is received from the server.

The mean response time for TB-TRT-issue is about one percent more than the TB-noTRT. The mean response time for TB-TRT-reuse is about fifty percent less than TB-noTRT. These results are shown in Figure 4.4.

These performance results show the potential for a server to increase its throughput and decrease its response time using a TRT when clients make repeat visits. With a TRT, a client can clearly re-establish trust with a server more efficiently compared to renegotiating trust from scratch.

An interesting question to consider from the results presented thus far is what

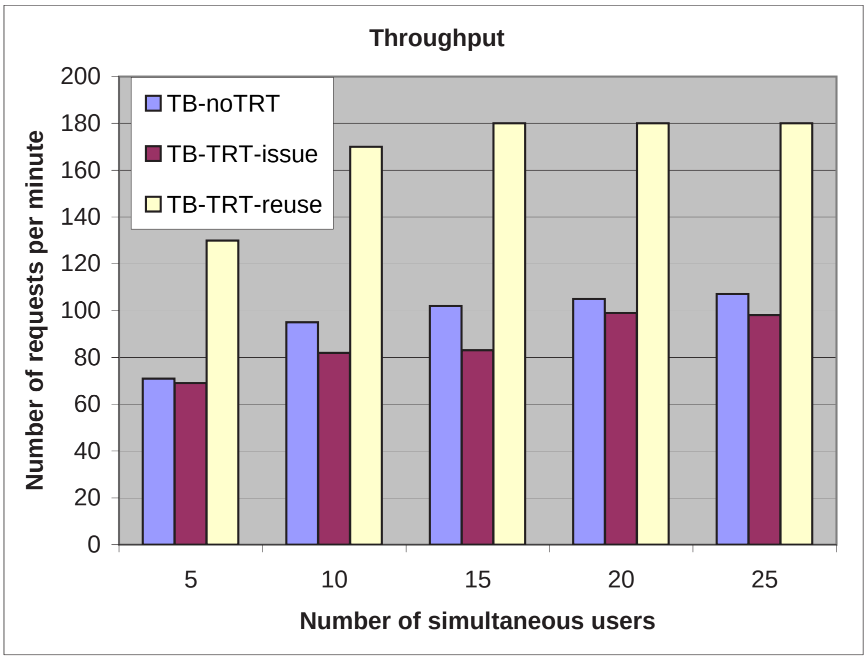


Figure 4.3: Server throughput during trust negotiations involving a TRT.

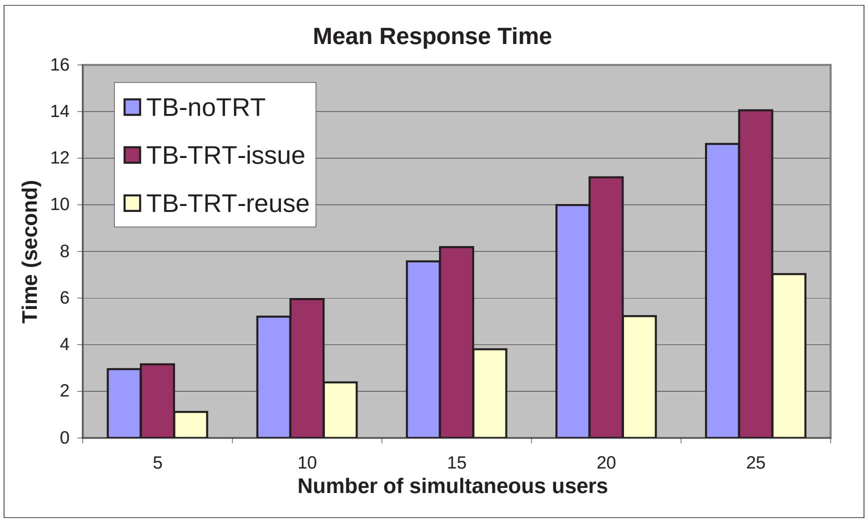


Figure 4.4: Mean response time for trust negotiations involving a TRT.

percentage of requests to a server need to be from repeat customers in order for the TRT to produce an overall savings. To answer this question, four use cases were constructed based on the earlier measurements to predict the performance of the system under various loads of repeat customers. The use cases assume that a server provides a number of protected resources, and the number of users ranges from 5 to 25. Some percentage of the users request a resource that requires the same level of trust that was negotiated during an earlier visit. Other users make initial requests that require trust be negotiated from scratch.

The first use case (TRT 0%) assumes that all users are first-time visitors. In this case, the server always performs the full traditional negotiation process and generates a new resumption token. Although a resumption token was generated and issued by the server during each request received by the server, none of the tokens were ever used in later requests. Thus, the resources that were used to generate the token were completely wasted.

The second use case (TRT 25%) assumes that 75% of the received requests are first-time requests from a user. The other 25% of the requests are repeat visits that require the same level of trust. The server performs a traditional negotiation process on 75% of the received requests and generates a new resumption token. For the other 25% of the requests, the server is able to provide the requested resource with the TRT the user submits.

The third use case (TRT 50%) assumes that half of the received requests are first-time requests from a user. The other half of the requests are repeat visits that require the same level of trust. The server performs a traditional negotiation process on half of the received requests and generates a new resumption token. For the other half of the requests, the server is able to provide the requested resource with the TRT the user submits.

The fourth use case (TRT 100%) assumes all the users are repeat customers that require the same level of trust originally negotiated. In this use case, the server only needs to perform the traditional negotiation process on users' first request. Thereafter, the server can provide the requested resource with the TRT the user submits.

Figure 4.5 compares these four use cases with TB-noTRT, where a negotiation is always performed regardless of the request pattern from its users. The comparison shows the cumulative cost in seconds for various test configurations involving 5, 15, or 25 users repeating a request from 1–5 times to a server. The cost is an estimate based on the average response time shown in Figure 4.4. The results indicate that the overhead of handling the TRT can be recouped when approximately 25% of the requests to a server are from repeat customers. The throughput and response time improve up to approximately 33% as the percentage of repeat customers grows to 100%.

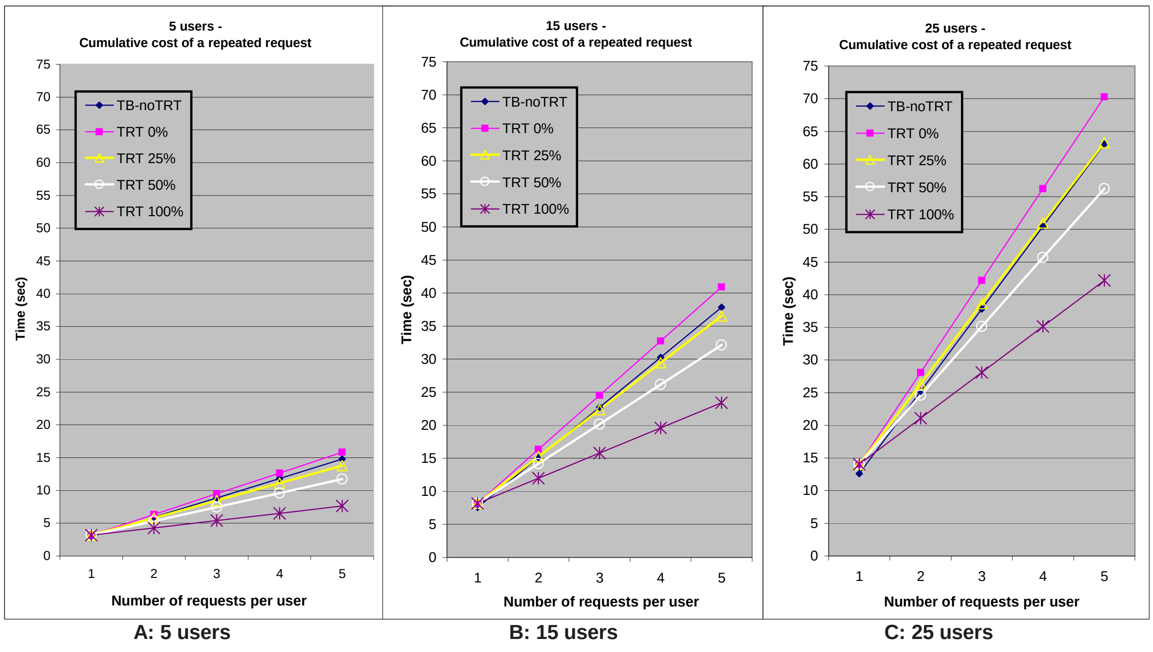


Figure 4.5: An estimate of the cumulative cost of a trust negotiation while varying the number of users, number of requests per user, and the percentage of repeat customers.

Chapter 5 — Related Work

This chapter describes work related to trust preservation.

5.1 Cookies

HTTP is a stateless protocol where each request is treated as a brand new request with no correlation to any prior request made by the same user. Cookies are a mechanism to preserve state between HTTP requests and responses. Typically, cookies are small files approximately 1KB in size [10].

A web server generates a cookie containing client state information and delivers it to a browser. If the browser is configured to accept cookies, the browser will store the cookie on the client's computer. Depending on the expiration date, a cookie remains on the client's hard drive after the browser is terminated. This processing is transparent to the user.

When a client reconnects to a server that previously issued a cookie to the client, the cookie is returned to the server as part of the HTTP request header, assuming the cookie has not expired. The server state is restored from the information contained in the cookie. Generally, the state information included in cookies are single sign-on information, user preferences, or the contents of a shopping cart [12].

Similar to cookies, the trust resumption tokens presented in this thesis are authenticators generated by a server and stored at the client. Trust resumption tokens can be implemented using cookies. However, this would limit the use of trust resumption tokens to the HTTP protocol. There are risks to storing authentication information on client computers. The trust resumption token presented in this thesis is designed to prevent well-known attacks against cookies [5], as discussed in Chapter 2.

5.2 Kerberos tickets

Kerberos tickets preserve authentication information. In Kerberos, the Authentication Server (AS) generates a ticket that preserves the client's authentication by creating a session key that is shared between the client and the TGS. The ticket is not attribute-based but provides evidence that the user has knowledge of the password or possesses the private key of the certificate that was used during authentication.

With password-based authentication and some certificate-based authentication, Kerberos requires a pre-existing relationship between a client and the AS before the client can access any services. In these cases, the Kerberos system administrator creates and manages a list of user accounts. In contrast, trust negotiation is used to establish trust between entities that have no prior relationship.

The design of trust resumption tokens presented in Chapter 2 has similarities to the Kerberos method of preserving the client's authentication. A TRT maintains the level of trust by creating a token containing the client's authentication information. The client presents this token to the server in order to re-authenticate to the same level of trust that was previously established. In Kerberos, the session key must be carefully protected by both the client and the TGT. In a system using the TRT, the private keys of the credentials whose attributes appear in the TRT must be protected, even though the TRT can be freely disclosed as long as the user is comfortable disclosing the attributes it contains. Mere possession of the credential is not sufficient to gain access to the server—the client must prove ownership of the credential. Unlike a Kerberos session key, these private keys are not shared with the other participating party. Section 2.2.1 discussed how this was done in more detail.

5.3 Trust negotiation in session level protocols

Transport Layer Security (TLS) is a session-layer protocol that establishes a secure connection between a client and a server. Authentication in TLS requires computationally expensive public-key operations [13]. In order to reduce the cost of a client and server re-authenticating each other, TLS supports session resumption. If a TLS server is configured to allow session resumption, the client is presented with a session identifier during the initial TLS handshake. When the session completes, the client and server store the session identifier and some cryptographic parameters relevant to the session. When the client returns to the server, the client presents the session identifier. If the server still remembers the session, the client and server use the cryptographic parameters from the previous session to establish new keying material to secure the new session. This avoids the costly public-key operations that were necessary to secure the first session.

Previous research in trust negotiation integrates trust negotiation into Transport Layer Security (TLS), but does not consider how the trust established during a session can be maintained [6]. Trust negotiation can be costly in terms of computation and number of number of rounds of communication. A recently completed thesis discusses how authentication information can be cached in the session-layer protocol to improve efficiency over the course of a session [8]. The thesis also discusses some issues related to caching trust during a session, demonstrates the necessity of maintaining the trust that is established during a session, and describes how trust can be cached in TLS [8].

Unlike the multi-session trust maintenance methods proposed in this thesis, TLS session resumption requires the server to maintain session state. Traditional trust negotiation may require a large number of public-key operations compared to traditional TLS authentication. When trust negotiation requires a large number of public-key operations, the use of a TRT during subsequent visits reduces the number of required public-key operations to a single credential validation and ownership verification.

5.4 Single sign-on

Single sign-on (SSO) is an approach whereby a user can authenticate once to access a protected resource and be granted access to other protected resources without the need to repeatedly enter a password. SSO automates user authentication processes by managing user passwords. The goal of SSO is to reduce human error and increase usability and efficiency.

Similar to SSO, trust negotiation is automated to reduce human error and increase usability. In contrast to SSO, which enables a principal to authenticate once and repeatedly access authorized resources, the current design of TrustBuilder does not provide any way to preserve a trust relationship beyond the current session except through renegotiation. In other words, trust is not persistent in the current model.

Preserving the trust established across multiple sessions is similar to many single sign-on paradigms like the Liberty Alliance Project [7] and Microsoft Passport [11]. In contrast to the Liberty Alliance Project and Microsoft Passport, trust negotiation enabled single sign-on is attribute-based. The alliance usage model for TRT presented in Section 2.2.2 permits the same authentication token to satisfy different access control policies across several servers that agree on a common set of attributes.

Chapter 6 — Conclusion and Future Work

In this thesis, two approaches to preserving trust across multiple sessions in open systems were presented. Previous work in trust negotiation was based on a simple model of per-request access control, and did not address the issue of preserving trust across sessions.

The first approach is to create a trust resumption token, which captures the details of a trust negotiation sufficient for trust to be efficiently resumed during a later session.

The second is the integration of existing systems, namely KeyNote and Kerberos, with trust negotiation to leverage their authenticator, the KeyNote assertions and the Kerberos ticketing service, for trust preservation. This integration experience provides a roadmap to make existing systems (KeyNote and Kerberos services) available to strangers outside their security domain (e.g., outside a Kerberos realm).

The contribution of this thesis is the application of ideas similar to the concept of single-sign-on in identity-based systems, but applied to the problem of authentication in open systems using trust negotiation.

Support for a trust resumption token was implemented in TrustBuilder, a prototype system for trust negotiation. The performance analysis in Chapter 5 demonstrates the potential of these approaches to improve the performance of trust negotiation compared to a system that does not preserve trust when there is a a modest amount of repeated access. Trust persistence promises to make authenticating strangers in open systems more efficient. Trust persistence also allows strangers to make friends—the relationship of trust persists because the parties involved are no longer unknown to each other. Having authenticated each other once, they do not need to perform the same expensive operations to resume the prior level of authentication.

The initial implementation of trust preservation in TrustBuilder stores attributes in the TRT. An interesting alternative to explore in the future is to store roles in the TRT. Both approaches offer advantages to applications. Storing attributes in the TRT reduces the risk of a client retaining invalid permission to access a resource when there is a change in the policy that maps attributes to a role. When policies are less likely change, storing a role in the TRT eliminates the need to repeatedly derive the role from the attributes when resources are accessed that require that role.

This thesis focused on enabling a client to resume trust with a server. If a client were to present a trust resumption token to a server, the server could easily regain trust established with the client. This approach would require a server to store a TRT for every client that accesses it. For high-volume servers, this could be a burden. Identifying techniques to reduce the burden on a server storing a TRT may be worth further exploration.

In a peer-to-peer scenario, the client-server role is often interchangeable between negotiation parties. The relationships or transactions between these parties are often transient; however, when the relationships are not transient, one might want to preserve the trust established via the negotiation process. Issues pertaining to trust preservation in a peer-to-peer scenario are also a topic for future research.

This thesis presents a high-level design of integrating TrustBuilder with Kerberos. An implementation of these ideas in Kerberos will extend its authentication facilities to strangers outside a Kerberos realm. In addition, the alliance usage model for TRT presented in Section 2.2.2 raises new opportunities for future work.

Bibliography

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System Version 2. In *Internet Draft RFC 2704*, 1999.

[2] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols Workshop*, Cambridge, UK, 1998.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996. IEEE Computer Society Press.

[4] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-7)*, pages 134–143. ACM Press, November 2000.

[5] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and dont's of client authentication on the web. In *10th USENIX Security Symposium*, August 2000.

[6] Adam Hess, Jared Jacobson, Hyrum Mills, Ryan Wamsley, Kent E. Seamons, and Bryan Smith. Advanced client/server authentication in TLS. In *Network and Distributed System Security Symposium*, pages 203–214, San Diego, CA, February 2002.

[7] http://www.projectliberty.org. *Liberty Alliance Project*, December 2002.

[8] Jared Jacobson. Trust negotiation in session-layer protocols. Master's thesis, Computer Science Department, Brigham Young University, July 2003.

[9] John Kohl and Clifford Neuman. RFC 1510: The Kerberos network authentication service (V5). IETF RFC, Sep 1993. Available from http://www.ietf.org-/rfc/rfc1510.txt.

[10] D. Kristol and L. Montulli. HTTP state management mechanism. IETF RFC 2965, October 2000.

[11] Microsoft. Microsoft .Net Passport. Microsoft Technical White Paper, Jan 2004. Available from http://download.microsoft.com/download/a/f/4/af49b391-086e-4aa2-a84b-ef6d916b2f08/passport_reviewguide.doc.

[12] Netscape. Persistent Client State HTTP Cookies. http://wp.netscape.com/newsref/std/cookie_spec.html, 1999.

[13] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[14] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[15] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume I, pages 88–102, Hilton Head, SC, January 2000. IEEE Press.

[16] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6):30–37, November/December 2002.