PROTECTING SENSITIVE CREDENTIAL CONTENT DURING TRUST NEGOTIATION

By

Ryan D. Jarvis

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

April 2003

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of the thesis submitted by

Ryan Dieter Jarvis

Each member of the following graduate committee has read this thesis and by majority vote found it to be satisfactory in its present form satisfying the thesis requirements for the degree of Master of Science within the Department of Computer Science at Brigham Young University.

Date	Dr. Kent E. Seamons, Chair
Date	Dr. Quinn Snell
Date	Dr. Aurel D. Cornell

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Ryan Dieter Jarvis in its final form and have found that (1) its format, citations, and bibliographic 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	David W. Embley Graduate Coordinator
Accepted for the College	G. Rex Bryce Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

PROTECTING SENSITIVE CREDENTIAL CONTENT DURING TRUST NEGOTIATION

Ryan D. Jarvis

Department of Computer Science

Master of Science

Keeping sensitive information private in a public world is a common concern to users of digital credentials. A digital credential may contain sensitive attributes certifying characteristics about its owner. X.509v3, the most widely used certificate standard, includes support for certificate extensions that make it possible to bind multiple attributes to a public key contained in the certificate. This feature, although convenient, potentially exploits the certificate holder's private information contained in the certificate. There are currently no privacy considerations in place to protect the disclosure of attributes in a certificate.

This thesis focuses on protecting sensitive credential content during trust negotiation and demonstrates, through design and implementation, the privacy benefits achieved through selective disclosure. Selective disclosure of credential content can be achieved using private attributes, a well-known technique that incorporates bit commitment within digital credentials. This technique has not been thoroughly explored or implemented in any prior work. In this thesis, a protocol for issuing and showing credentials containing private attributes is discussed and suggested as a method for concealing and selectively revealing sensitive attributes bound to credentials during trust negotiation. To demonstrate greater privacy control within a credential-based system, private attributes are incorporated into TrustBuilder, an implementation of trust negotiation. With access control at the attribute level, TrustBuilder gives users greater control over their private information and can improve the success rate of negotiations. TrustBuilder also demonstrates how credentials with private attributes can eliminate risks normally associated with exchanging credentials, such as excessive gathering of information that is not germane to the transaction and inadvertently disclosing the value of a sensitive credential attribute.

ACKNOWLEDGEMENTS

I would like to thank my Father in Heaven for answering my prayers and helping me find my way in becoming who I am. I also would like to thank my wife, Tonya, for all her wonderful support getting me through school successfully, my professor, Dr. Kent Seamons, for hiring me as a research assistant in his Internet Security Research Lab at BYU, and all my colleagues at school for assisting me throughout the journey. My research was supported by DARPA through AFRL contract number F33615-01-C-0336 and through Space and Naval Warfare Systems Center San Diego grant number N66001-01-18908.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1: INTRODUCTION

Alice is an online shopper ready with her cart of items to check out from Bob's online store. Bob requires that Alice send him a digital credential containing her name and email address before he will finish processing her online purchase. Alice agrees, but the only credential she has with her name and email address also contains her social security number and weight – attributes she is not willing to reveal to Bob. If Alice were to disclose only her name and email address, Bob would not be able to determine if they were certified by a trusted CA. Alice would prefer a method to let Bob have access to the attributes he requires within a credential without having to disclose additional, irrelevant, sensitive information in the same credential. Figure 1 illustrates the idea of Alice being able to selectively disclose attributes in her credential while keeping sensitive attributes private.

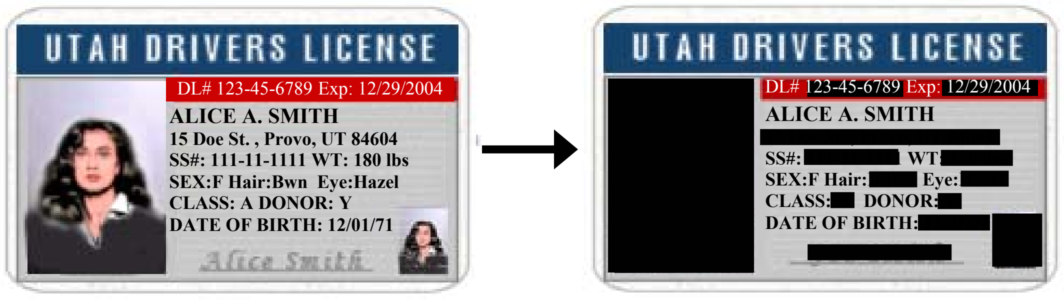


Figure 1 – Selective Disclosure. Alice A. Smith's digital credential contains sensitive attributes that Alice does not want to make public. Selective disclosure allows Alice to control access to her attributes, and thereby protect the disclosure of her personal information.

A digital credential is the on-line analogue to the physical credentials that people carry in their wallets. A credential contains attributes about the credential owner asserted by the credential issuer, usually represented as name/value pairs. It is signed by the issuer's private key, and can be verified using the issuer's public key. A credential may contain the public key of the owner, permitting the owner to use the corresponding private key to answer challenges or otherwise demonstrate ownership of the credential. Credential is a more general term for certificate. The most widely used certificate standard is X.509v3, which supports an extension mechanism used to bind additional attributes to the certificate owner within a single certificate. Unfortunately, this extension mechanism is commonly used without privacy considerations and poses a threat to certificates becoming convenient dossier builders.

Currently, only a few applications treat credentials as potentially sensitive. Westin [25] defines privacy as "The right of individuals to determine for themselves when, how and to what extent information about them is communicated to others." An example of the potential threat to personal privacy is demonstrated in [15] where VeriSign, a certificate distribution company, is shown to have recently issued public credentials containing sensitive demographics to an estimated three million users worldwide.

1.1. Thesis Statement

This thesis focuses on the problems associated with protecting a user's sensitive credential content during trust negotiation and demonstrates, through design and implementation, the privacy benefits warranted through selective disclosure. Selective disclosure of credential content can be achieved using private attributes, a well-known technique based on *bit commitment* [10], where a value may be committed to without being immediately disclosed, but in the context of digital credentials. A protocol for issuing and showing credentials containing private attributes is discussed and suggested as a method for concealing and selectively revealing sensitive attributes bound to credentials during trust negotiation. To provide greater privacy control within a credential-based system, private attribute support is incorporated into TrustBuilder, an implementation of trust negotiation. With finer-grained disclosure policies, the new TrustBuilder gives users greater control over their private information without requiring additional real-time user interaction. In addition, the new TrustBuilder demonstrates how credentials with private attributes can help successfully establish trust during trust negotiation as well as eliminate certain risks normally associated with exchanging credentials.

1.2. Organization of Thesis

The remainder of this thesis is organized as follows. In Chapter 2, the foundation material our research relies upon is introduced. Chapter 3 presents the design goals for selective disclosure of credential content and an analysis of the alternative approaches. In Chapter 4, the design and implementation incorporating selective disclosure of credential content within trust negotiation for enhanced privacy protection is presented. Chapter 5 gives concrete examples of how selective disclosure benefits trust negotiation and credential based systems, Chapter 6 contains the related works, and Chapter 7 finishes the thesis with the conclusions and future work.

CHAPTER 2: FOUNDATION MATERIAL

This section introduces the primitives that are important to protecting digital information in open systems, including secure one-way functions, public key cryptography, digital signatures, digital credentials, private attributes, trust negotiation and an implementation of trust negotiation called TrustBuilder.

2.1. Cryptographically Secure One-Way Functions

Cryptographic primitives are important to protecting digital information. The one-way function is unique because of its mathematical nature of only being computable in a single direction. In other words, with a strong one-way hash function it is computationally infeasible to determine the original value given the ending result or determine another value that will end up with the same result as the original. The result, or *digest*, can act as a unique fingerprint for identifying the input without divulging the input value. Digests are useful for proving ownership of digital information, representing large amounts of data with a small fixed output and for being able to make commitments to sensitive information.

The two most widely used standards for secure one-way hash functions are the message-digest algorithm (MD5) and the secure hash algorithm (SHA-1). MD5 was published as RFC 1321 in 1992 by Ron Rivest [17] and was the most widely used secure hash function until recent brute force and cryptanalytic concerns arose. The MD5 algorithm takes as input a message of arbitrary length and produces a 128-bit message digest, processed in 512-bit blocks. SHA-1 was developed by the National Institute of Standards and Technology (NIST) with its latest version FIPS PUB 180-1 published in

1995 [11]. The SHA-1 algorithm takes as input a message with maximum length 2^{64} and produces a 160 bit message digest processed using 512-bit blocks as well.

2.2. Public Key Encryption and Digital Signatures

Public key cryptography was initially proposed by Diffie and Hellman in 1976 [3] and was a complete revolution in how encryption could be performed. The new system was based on mathematical functions and operated using two separate asymmetric keys, a private key and a public key. Unlike prior encryption schemes where the same key was used to encrypt and decrypt, public key cryptography uses one key to encrypt and the other key to decrypt. In the most widely used implementation of public key cryptography, RSA (Rivest, Shamir and Adleman) 1978 [16], the encryption and decryption keys can be used interchangeably. What the private key encrypts, only the public key decrypts and what the public key encrypts only the private key can decrypt. This feature is very useful for achieving certain aspects of confidentiality and authentication.

A high level example of using public-key cryptography is as follows: Bob takes his plain text love letter message and feeds it into the encryption algorithm using his girlfriend Alice's public key. The resulting cipher text is then sent from Bob to Alice. Alice, upon receiving the cipher text feeds it into her decryption algorithm with her private key. The resulting output is the original plaintext love letter, which only Alice can enjoy.

If Alice needs to be sure that the letter comes from Bob, he can send the same love letter again but also encrypted with his private key. When Alice receives the message the second time, she feeds the cipher text along with Bob's public key into the decryption algorithm and then takes that output and feeds it again into the decryption algorithm using her private key. If the output is the love letter, she knows it must be from Bob and only she can read it. These examples show how public key cryptography can be used to provide confidentiality as well as mutual authentication.

Private keys are often used to sign document digests to provide authentication and integrity. If Bob really wanted Alice to know that the love letter was from him and it was legitimate, he could have used his private key to sign a digest of the letter and send the letter and the signed digest to Alice. This is similar to techniques used to create a message authentication code (MAC) to validate the integrity of documents. Typically signing the digest of a document with a private key creates what is called a digital signature. Any person can verify the digital signature of a document by first creating its original digest by feeding the document minus the signature into the same secure hash function used, such as MD5 or SHA-1, decrypting the document's signature with the corresponding public key and comparing the two plaintext digests. If they match, the signature is verified and the document has integrity. Achieving the properties of integrity and authentication are crucial for the success of protecting sensitive credential content during trust negotiation.

2.3. Digital Credentials

Digital credentials are best described when compared to the physical credentials people have in their wallets. Like a driver's license, trusted third parties who certify specific attributes about entities, or credential owners, issue the physical credentials we carry in our wallets. In the case of a physical driver's license, the Department of Motor

Vehicles (DMV) acts as the trusted third party and certifies our name, age, eye color, weight and possibly a social security number and vehicle restrictions.

The most universally accepted standard for digital credentials is currently the public-key certificate format: X.509v3. The initial version of X.509 was published in 1988, version 2 was published in 1993, and version 3 finally approved as a standard in 1995 [9]. X.509v3 certificates contain the following fields [24]:

- version
- serial number
- signature algorithm identifier
- issuer name
- validity period
- subject name
- subject public key information
- issuer unique identifier
- subject unique identifier
- extensions
- a signature on the digest created from all of the prior fields combined

Extensions are basically additional attribute placeholders in a certificate. It is possible to have zero or more extensions and they can vary in size and number.

X.509v3 relies upon the association of public key certificates with users. The certificates or credentials are assumed created by a trusted third party, known as a Certificate Authority (CA). The certificate contains the public key of the subject, while the corresponding private key is kept secret by the certificate owner and can be used for authentication and encryption. Certificates are typically revoked when a private key is compromised by subjects or issuers. A list of certificates that are revoked is called a certificate revocation list, and can be used and distributed to stop invalid certificates from being accepted.

Certificates can also be constructed into chains denoting a hierarchical relationship such as ACCREDITING BODY → BYU → STUDENT. In a certificate chain each certificate's signature can be verified using the public key of the parent certificate. To completely verify the chain, the verifier will only need to have the public key of the root, which in the last example was the accrediting body. This is a nice feature giving verifiers the convenience of only having to store possible root keys instead of all public keys needed for the complete verification of certificates. Certificate chains are very useful when negotiating trust.

2.4. Bit Commitments and Private Attributes

One design for creating attributes that can be selectively revealed as part of a credential borrows directly from the idea of bit commitment [10] [18], which allows a user to commit to a value without having to immediately reveal it. Thus, when sensitive attributes in a credential are replaced with commitments to those values, we refer to those commitments as *private attributes.*

Although the idea of using bit commitment within digital credentials is not new, the approach has never been explored or implemented. Both Brands and Renfro make observations about the technique. Brands [2] observed, "Another attempt to protect privacy is for the CA to digitally sign (salted) one-way hashes of attributes…" and Renfro [15] mentioned, "One approach is to append a nonce to the information in the database and including the hash of the information and the nonce in the end-entity certificate". Both observations allude to the simple well-known technique we call private attributes. The following issuing protocol describes an approach for issuing a credential containing private attributes. It differentiates itself from the bit commitment protocol given in [18] by using only a single random value and being used within the context of digital credentials. There are minor variations to this basic protocol, but the protocol presented is sufficient to demonstrate the general approach.

Suppose Alice requests that a credential be issued to her by a certifying authority (CA). The typical arrangement is for Alice to construct a credential request that contains the information to be included in the credential. Alice sends her request to a CA, who verifies her sensitive attributes and returns a signed credential to Alice.

In the issuing protocol, Alice creates a credential request to be signed by the CA. She will complete the following four steps for each sensitive attribute she wishes to remain private within her credential.

1. Alice generates a random-bit string r.
2. Alice creates a pre-image p consisting of the sensitive value v she wishes to remain private concatenated with r.

   $p = v \mid r$

3. Alice generates a private attribute a_1 by invoking a collision-resistant, one-way function on the pre-image.

   $a_1 = \text{oneway}(p)$

4. The commitment a_1 to Alice's sensitive value v is her private attribute to be included as part of her credential request and stored in the signed credential in place of v.

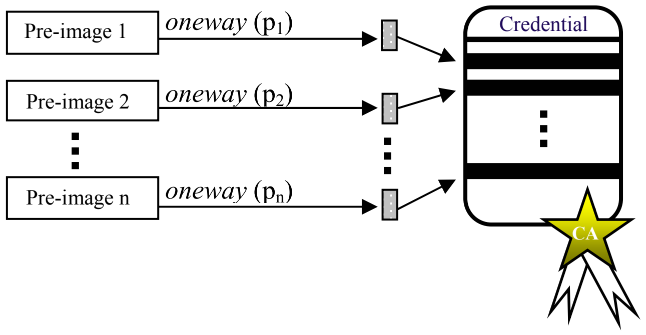


Figure 2 – A Credential containing Private Attributes being issued. Using a one-way-function, bit commitments are generated from pre-images, p_1 … p_n, and placed into a credential request. The digests represent commitments to sensitive attributes whose values cannot be determined without the corresponding pre-images.

Alice submits the credential request to the CA, along with her corresponding pre-images. The CA must ensure the pre-images are verified and that they correspond to the commitments within the request.

The following is an example of a verification protocol, also shown in Figure 3, which allows Alice to disclose private attributes selectively in her credential to Bob.

1. Alice sends Bob her digital credential that contains the private attribute a_1, and Bob verifies the credential.
2. When Alice determines that Bob is authorized to receive the value she committed to, Alice sends Bob the original pre-image that contains the sensitive value v and random-bit string r.

   $p = v \mid r$

3. Bob calls a collision-resistant, one-way function on the pre-image to generate commitment a_2 using the same function that was used during the issuing protocol.

   $a_2 = \text{oneway}(p)$

4. Bob compares a_2 with the private attribute contained within the credential. If they match, Bob accepts the value v as Alice's legitimate attribute value.

valid if $(a_1 == a_2)$

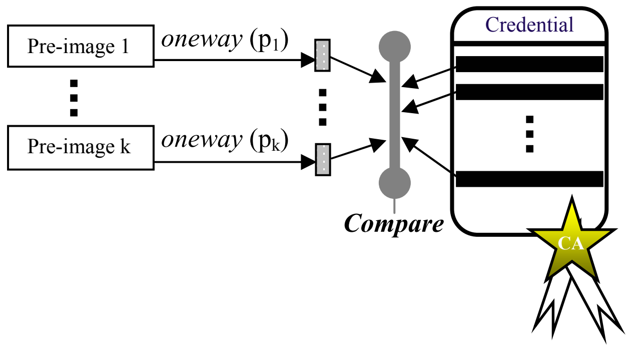


Figure 3 – The Private Attribute Verification Process. Taking the pre-images sent $p_1 \ldots p_k$, calling the one-way function on them, and comparing the result to the corresponding commitment within the credential verify a credential's attributes.

This verification protocol demonstrates how Alice can prove to Bob, the verifier, the original values of selected private attributes in her credential, as shown in Figure 3.

2.5. Trust Negotiation

Private attributes can enhance privacy protection within credential-based online applications. One application area that can benefit from private attributes is trust negotiation, a new approach to establishing trust between strangers through the process of iteratively disclosing digital credentials that describe attributes of the negotiation participants [28]. This approach relies on access-control policies that govern access to protected resources by specifying credential combinations that must be submitted to obtain authorization.

A digital credential may contain sensitive information and its disclosure must be carefully managed in accordance with an access-control policy that specifies which credentials must be received before it can be disclosed. A credential is disclosed when its access-control policy has been satisfied. Disclosure policies may govern access to all sensitive resources, including credentials, roles, capabilities, policies, and services.

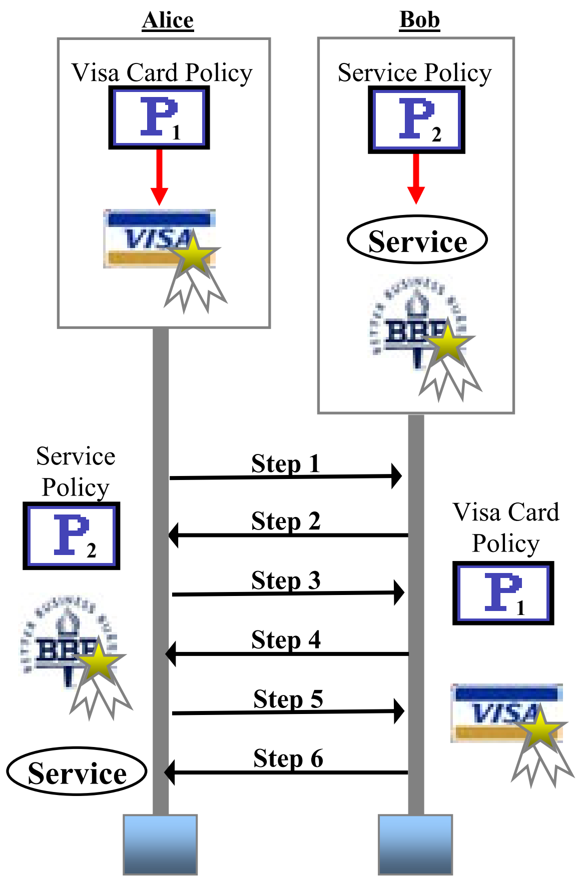


Figure 4 – Trust Negotiation Example. In this example, strangers Alice and Bob perform a trust negotiation in which Alice obtains a service from Bob after exchanging policies and credentials. The six steps are as follows: (1) Alice requests service from Bob, (2) Bob discloses his policy P2, (3) Alice discloses her policy P_1, (4) Bob discloses his BBB credential, (5) Alice discloses her Visa card credential, (6) Bob grants service.

Figure 4 illustrates the following example of a trust negotiation.   Alice wants a service from a stranger, Bob, but Bob requires that she send him her Visa card credential to access the service. Bob discloses an access-control policy to Alice to inform her of his requirements to access the service. Alice will disclose her Visa card only to members of the Better Business Bureau (BBB). She discloses her policy to Bob, who responds with his BBB credential that he is willing to share with anyone. Next, Alice discloses her Visa credential. Since Alice is now authorized to receive the service, Bob grants her access. Negotiations are currently based on access control decisions at the credential level. Protecting sensitive credential content during trust negotiation, which provides finer grained access control at the attribute level, will be discussed later in Chapter 4.

2.6. TrustBuilder

Our current implementation of trust negotiation uses the TrustBuilder architecture [1] [28] to conduct trust negotiation between security agents that mediate access to protected resources, services, access-control policies, and credentials. The trust-negotiation message protocol describes the ordering of messages and the type of information the messages contain. The trust negotiation strategy is the mechanism that controls the exact content of the messages, i.e., which credentials to disclose, when to disclose them, and when to terminate a negotiation.

The TrustBuilder architecture is composed of three main components, as depicted in Figure 5. The credential verification module checks the validity of each received credential, verifies each credential signature, examines possible revocation, discovers any credential chains, and answers challenges that require demonstrating credential ownership. The policy compliance checker ensures that a local resource is disclosed only to the other party after its policy has been satisfied and determines which local policies are satisfied by the other party's disclosed credentials. Finally, the negotiation-strategy module takes a description of the current stage of the negotiation, including the party's local credentials and policies and all the credentials and policies disclosed in previous rounds of the negotiation, and produce the next message to be sent to the other party.

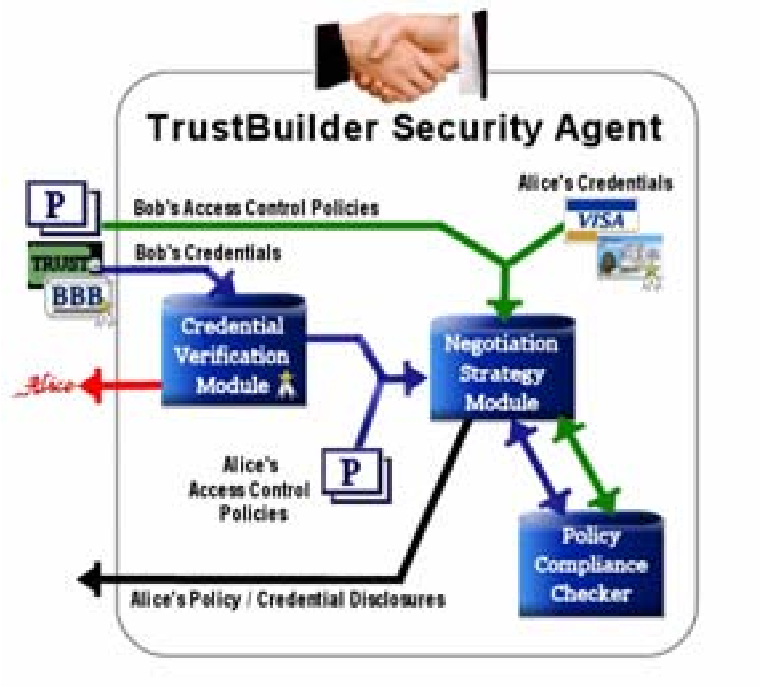

TrustBuilder Security Agent
Bob's Access Control Policies
Bob's Credentials
Alice's Credentials
Credential Verification Module
Negotiation Strategy Module
Alice's Access Control Policies
Policy Compliance Checker
Alice's Policy / Credential Disclosures

Figure 5 – TrustBuilder. A representation of the TrustBuilder architecture mediating access to credentials, access-control policies and services is depicted.

Access-control policies for local resources indicate which credentials are required before access is granted. Each TrustBuilder agent is equipped with a policy-compliance checker used for two main purposes. The first is for checking which incoming remote credentials satisfy local policies. The second is for determining which local credentials satisfy incoming remote policies.

Examples from our recent research in trust negotiation include support for sensitive credentials and access-control policies [22], the definition and interoperability of trust negotiation strategies [31], a trust-negotiation protocol based on an extension to TLS [7], protecting privacy during trust negotiation [21], an analysis of policy languages for trust negotiation [20], and the development of the TrustBuilder architecture for trust negotiation [28].

TrustBuilder provides support for sensitive credentials. Extending TrustBuilder to support private attributes would allow support for selective disclosure of credential content in trust negotiation. This fine-grained access-control would require that access-control policies be associated with attributes of a credential instead of at the credential level, as illustrated in Figure 6.

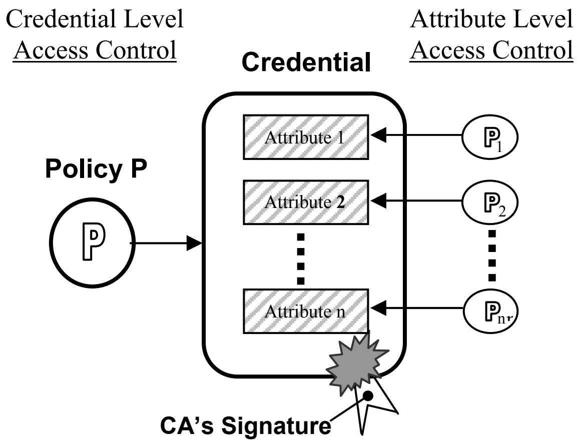


Figure 6 – Levels of Access Control. Policy P is an access-control policy that protects a sensitive credential, and constitutes credential level access-control. P_1 – P_n are policies independently govern access to each attribute and constitute attribute-level access control.

CHAPTER 3: SELECTIVE DISCLOSURE OF CREDENTIAL CONTENT

This chapter presents the design goals for selective disclosure of credential content and provides an in depth analysis of how private attributes can satisfy these goals.

3.1. Selective Disclosure Design Goals

This section presents general design goals for selective disclosure of credential content. These goals will serve as a basis for evaluating different approaches to safeguarding the privacy of credential content.

A credential that supports private attributes must exhibit:

1. **Confidentiality** – It is computationally infeasible to determine any information about a private attribute from the credential itself.
2. **Integrity** – When the owner of the credential decides to reveal the actual value of a private attribute, the owner must have to ability to convince the relying party the value extracted from the private attribute pre-image is indeed the same value that was certified by the CA.
3. **Access Control** – Private attributes in a credential are disclosed only at the credential owner's discretion.
4. **Performance** – The computational overhead required for issuing and verifying the credential's private attributes is not prohibitive.
5. **Usability** – It is feasible to integrate private attributes into existing credential standards and systems that make use of credentials.

3.2. Analysis of Private Attributes

The **confidentiality** of the credential's sensitive attributes is protected by the strength of the one-way function and the construction of the pre-image. Appending a random bit string to the attribute value (depicted in Figure 7) creates the pre-image. Using a fixed-length random bit string of 128 bits is sufficient to discourage dictionary attacks.

If random data are not appended to an attribute value, the attacker can iterate through the range of possible attribute values to find a match. For example, if the Social Security Number of the credential owner were stored in the credential as a private attribute without any random data appended to the value, an attacker who has access to the credential could simply iterate through all 10^9 possible social security numbers, invoking the one-way function on the potential value to generate a commitment, and determine the owner's social security number whenever the resulting commitment matches the commitment stored in the credential. Appending random data of 128 bits would increase the cost of an attack to 2^{128} iterations for every possible attribute value, a prohibitive amount for any attacker.

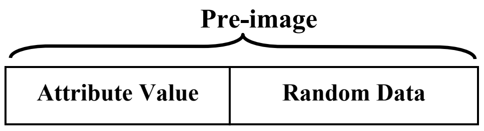


Figure 7 – The Pre-image. When creating the pre-image to appropriately protect against dictionary attacks, random data should be appended to the actual attribute value.

The **integrity** of each attribute committed to within the credential is maintained by the properties of the one-way function. In our case, the secure hash algorithm version 1 (SHA1) [24] is used, which takes a message of less than 2^{64} bits in length and produces a 160-bit message digest. It is computationally infeasible to find collisions using a strong one-way function. Thus it is highly improbable that a credential owner would be able to generate an alternative pre-image with a different value than the CA certified and successfully present it to a relying party.

The design of the pre-image must insure that there is no ambiguity between the attribute value v and the random string r. For instance, either the random bit string could have a predetermined fixed length, or there must be a distinct delimiter to accurately distinguish between v and r.

The credential owner can enforce **access control** over sensitive credential attributes by keeping the associated pre-images secure and disclosing them only to authorized parties that are entitled to know the contents of a private attribute. This protection has limits, because once an authorized entity has access to the pre-images, there is no mechanism in the credential to prevent that entity from further disclosing the private attribute. In addition, the verification protocol must be confidential in order to prevent an eavesdropper from determining the private attribute. Incorporating private attributes in trust negotiation permits the credential owner to maintain direct control of pre-images, without requiring that personal information be stored in a centralized database.

The **performance** of private attributes can be analyzed in terms of both computational overhead and storage requirements. In terms of computational overhead, one-way hash functions such as SHA1 introduce minor computational demands when compared to the much more expensive public key cryptography operations that are required to sign a credential.

The storage overhead for private attributes is minimal, requiring the replacement of the attribute value with a 20-byte commitment when using SHA-1. The storage requirements of the digital signature in the credential are significantly greater than the cost of a single private attribute. In some cases, a commitment may significantly reduce the size of a credential if a large attribute value, such as an image or biometric data, should be stored in the credential. If the attribute is not always required when the credential is in use, the space savings could provide motivation for a commitment, independent of the privacy advantages.

In cases where multiple private attributes are stored in a credential and the space requirements become an issue, it is also possible to represent multiple sensitive attributes with a single commitment using Merkle hash trees [8], which can significantly reduce the overall size of the credential by representing all private attributes with a single commitment.

To create a Merkle hash tree, each attribute is processed through a one-way function creating digests that will be the leaf nodes of the tree. Because the hash tree is a binary tree, meaning each parent node has only two child nodes, every two leaf nodes are combined, their values concatenated together, and processed through a one-way function to create the parent. This process continues as a binary tree is built and a single root result is used to represent all the attributes in the tree. It is then possible to disclose selectively an attribute by disclosing the attribute and necessary one-way function results in the tree to be used together for computing the root result and compare it to the root that could be stored in the certificate.

The issuing and verification steps for supporting hash trees differ slightly from the protocols introduced in this paper. The number of hashes required to create a hash tree given n attributes is 2n, and the number of hashes required for verification is the depth of the binary hash tree. The private attribute protocol presented earlier only requires one hash for creation and a single pre-image for verification.

The **usability** of private attributes will be discussed in detail in Chapter 5. We show that private attributes can be incorporated into X.509v3 certificates. A discussion of how existing systems can be easily modified to support credentials that contain private attributes is also presented.

CHAPTER 4: THE TRUST NEGOTIATION IMPLEMENTATION

Selective disclosure of credential content empowers individuals to determine for themselves when, how and to what extent information about them is communicated to others. By incorporating selective disclosure into TrustBuilder, users can have control over disclosing their credential's sensitive content in an automated fashion. The following six changes are required to support private attributes within a credential based system, such as TrustBuilder.

1. Incorporating private attributes within standard digital credentials.
2. Providing a network message protocol for transporting private attribute pre-images independent of their credentials.
3. Modifying policy compliance systems to verify private attributes and their corresponding pre-images.
4. Associating disclosure policies with private attribute pre-images.
5. Creating a negotiation strategy that provides attribute level access control by selectively disclosing sensitive private attribute pre-images.

The following sections in this chapter will detail the design and implementation of these required modifications within TrustBuilder and its support infrastructure.

4.1. Private Attributes within Standard Credentials

X.509v3 is the most widely used certificate standard. X509v3 certificate extensions [14] make it possible to bind multiple attributes to a single certificate or public key, which, although convenient, can potentially exploit the certificate holder's sensitive information in the certificate. There are currently no privacy considerations within

X.509v3 that protect the disclosure of attributes within a certificate. All attributes in a certificate are as public as the certificate. Whenever a certificate is disclosed, so are its sensitive attributes.

To counter this privacy vulnerability, any sensitive attribute normally embedded in an X.509v3 certificate can be alternatively replaced with a private attribute committing to the sensitive information (see section 2.4). Although additional overhead is required to handle private attributes within digital credentials, simply storing private attributes within X.509v3 certificates is straightforward.

```
Extension  = SEQUENCE  {
    extnID         OBJECT IDENTIFIER,
    critical       BOOLEAN DEFAULT FALSE,
    extnValue      OCTET STRING  }
```

Figure 8 – X.509v3 Extension Composition. X.509v3 extensions are ASN.1 structures - a sequence with three fields: extnID - the object identifier, critical bit - a boolean value, and extnValue - an octet string.

X.509v3 extensions are ASN.1 structures, each containing an object identifier (OID), critical bit, and octet string as shown in Figure 8. Normally an attribute value is placed into the extnValue field as an octet string defined by its OID. An OID is used to identify the content of the extension associating it with a type or a name allowing the ASN.1 octet string value to be interpreted correctly. There are pre-defined public OIDs recognized by most systems and custom internal OIDs only recognized by some systems. In order for most applications to recognize a private attribute, a new OID could be registered to represent a private attribute version of an already well-established type. For the TrustBuilder implementation, custom OIDs were chosen for identifying private attributes. In the future we may chose to register private attribute OIDs for general use.

As far as being able to store private attributes within X.509v3 certificates, storing private attributes is not any different than storing normal attributes within the extensions of a certificate. When a certificate extension is used to store a private attribute, the OID used must be understood by the application. The application can then verify and process it appropriately.

The critical bit within an X.509v3 extension identifies it as critical or not. If it is set to critical, it must be recognized or the certificate is invalid. The X.509 certificate standard RFC 3280 [9] states that a certificate-using system must reject a certificate if it encounters a critical extension that it does not recognize. According to the specification, the critical bit can be set to false when the extension may not be recognized, such as the case with a private attribute. By setting the critical bit to false, credentials containing private attributes are backwards compatible and interoperable with existing credential based systems.

No modifications are necessary to embed private attributes in X.509v3 certificates, but in order to appropriately issue certificates containing private attributes several additional steps are mandatory to verify the certificate. One possible approach for issuing credentials containing private attributes is described in section 2.4. Using this approach, the CA, in addition to the normal procedures performed when a CA verifies and signs a certificate request, verifies each private attribute digest stored in the certificate.

Typically the certificate requestor sends a certificate request to the CA that already contains the attribute data in the correct fields and only needs to be verified and signed by the CA. If the CA were to accept this approach when issuing certificates with private attributes, the CA would receive a certificate request with the private attribute digests already embedded. To verify that the digests in the certificate request correspond correctly to valid pre-images, the certificate requestor would send the original pre-images to the CA. To complete the verification, the CA would hash each pre-image and compare each digest with the private attribute digest already embedded within the certificate request as shown in Figure 9. If all digests match, and all extracted values from the pre-images of the certificate are appropriate, the CA will sign the certificate and return it to the certificate requestor.

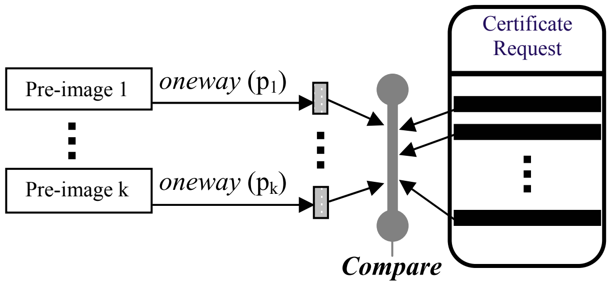


Figure 9 – The CA's Verification Process. Taking the pre-images sent by the certificate requestor $p_1 \ldots p_k$, the CA calls the specified one-way function on them and compares each result to the corresponding private attribute digest within the credential verifying the credential's sensitive attributes.

4.2. Creating and Communicating Private Attributes

Creating private attribute digests and pre-images needed for certificate requests can be completed with ease using Java's security library. A program was developed that takes the original plaintext attribute value, given as an argument or filename, and outputs two text files, one containing the private attribute pre-image and the other containing the Base64 encoded private attribute digest. The Java code responsible for creating the private attribute pre-image is shown in Figure 10. The entire Java source code is in section 8.2.1 of the appendix.

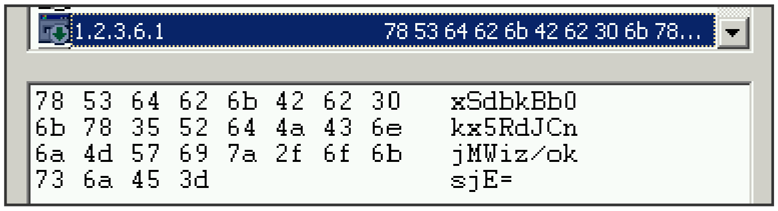
```
//Create Random String for Pre-image
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
random.setSeed((new Date().getTime()));
byte bytes[] = new byte[20];
random.nextBytes(bytes);
String randomStr = new String(bytes);
randomStr = new String(Base64.encode(bytes));

//Construct the Pre-image
String preimage = Value + ":" + randomStr.substring(0, 20);
System.out.println("Preimage created: " + preimage);
```

Figure 10 – Private Attribute Pre-image Creation. A secure random string is generated, Base64 encoded, and then truncated. The resulting 20 bytes of random data is appended to the original attribute value. The strength of the pre-image is in the random data.

The pre-image text is placed into a file associated with the credential and protected by the operating system. The file is an ASCII-based text file named after the credential and formatted with a single name-value pair per line delimited by a tab character representing the OID (name) to pre-image (value) relationship. Multiple pre-images can then be declared for a single credential. The associated digests are placed as extensions in the certificate request. An example of the pre-image and its associated certificate extension digest is shown in Figure 11.

PREIMAGE:	SecretAgent:3QeJFkR6VDYVWy2ws0gEdBMrjh8=



```
1.2.3.6.1                78 53 64 62 6b 42 62 30 6b 78...

78 53 64 62 6b 42 62 30    xSdbkBb0
6b 78 35 52 64 4a 43 6e    kx5RdJCn
6a 4d 57 69 7a 2f 6f 6b    jMWiz/ok
73 6a 45 3d                sjE=
```

Figure 11 – A View of the Private Attribute Pre-image and its associated digest. The example certificate's extension contains a private attribute digest visible in the Windows Certificate Viewer. The associated pre-image used to originally create the digest contains the actual attribute value "SecretAgent" followed by a separator and 28 bytes of random data.

Once a certificate contains private attributes, the associated pre-images will need to be handled appropriately during the showing protocol. In order to maintain the relationship between the certificate extension and its pre-image, a simple network message protocol can be used when transporting pre-images. Each pre-image message can be a printable string containing a unique name, OID, and pre-image data with each separated by a colon, and ending with a semicolon. In the implementation, the unique name used is the Base64 encoded MD5 hash of the certificate, the OID is the number string in ASCII typically used to denote the certificate extension, and the pre-image data is the actual ASCII pre-image. This format provides a unique mapping that allows the pre-images to be transmitted independent of their certificate.

One standard message protocol to exchange messages between a client and server on the internet is HTTP, or HTTPS. TrustBuilder currently runs over HTTP(S) with both the client and server independently communicating with their TrustBuilder Agents using SOAP as shown in Figure 12. The necessary modifications that were made to

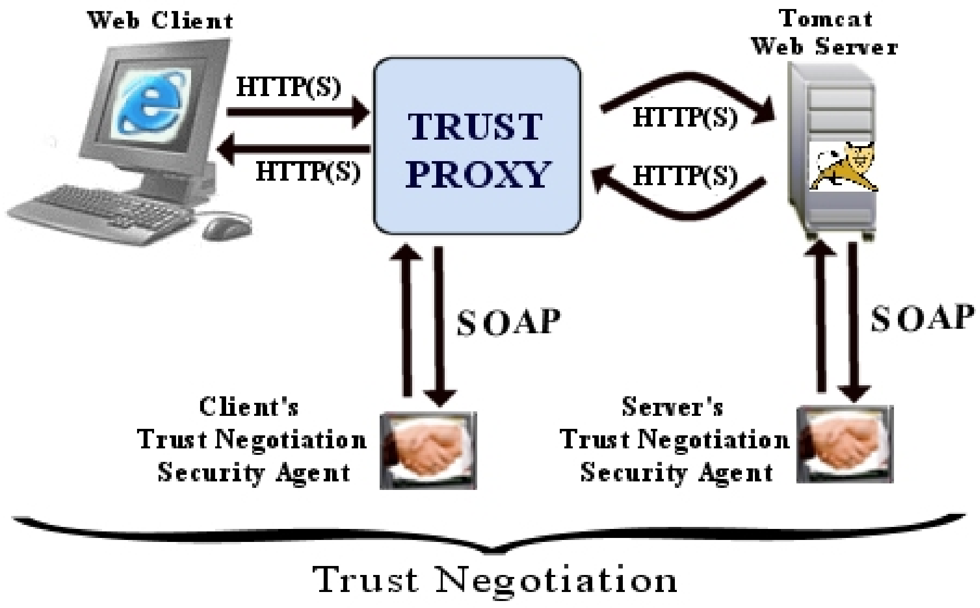


Figure 12 – The TrustBuilder Implementation Architecture. The web client's HTTP requests are handled through a trust proxy supporting trust negotiation. SOAP is used to communicate from the application layers to the independent security agents, and all other messages between the client and the server are over HTTP(S).

communicate private attribute pre-images included: adding a HTTP header called "PREIMAGES=", creating methods to extract pre-images from the HTTP header, and methods to construct the HTTP header with the private attribute pre-image messages. The existing SOAP communication only required minor modifications for it to include a vector of pre-image message strings in the TrustBuilder method call.

4.3. Policy Compliance and Private Attribute Verification

In trust negotiation, policy compliance is viewed as a black box that basically takes a policy and a set of credentials as input and outputs a boolean value indicating whether or not the policy is satisfied. Optionally, it may also output the set of credentials that satisfy the policy. The compliance checker is critical for determining if a policy can be satisfied and which credentials are relevant to the negotiation. See [20] for a complete list of features recommended in a compliance checker for trust negotiation.

TrustBuilder supports the IBM Trust Establishment (TE) system. TE utilizes the Trust Policy Language (TPL) developed by IBM for specifying policies that govern resources and credentials. Example policies can be found in section 8.1.3 of the appendix. TE also supports standard X.509v3 certificates as its credential format. TE has a proprietary storage file format for the X.509v3 certificate private keys and credential storage database, and requires several nonstandard certificate extensions to be included in each X.509v3 certificate for valid use with its compliance checker.

TrustBuilder currently uses the TE policy compliance checker to determine whether or not credentials satisfy a policy. Because the TE system is not open source, we were unable to incorporate support for private attributes directly into the compliance checker. If it were possible, we would modify the TE compliance checker to take additional arguments, such as the necessary pre-images needed to validate private attributes and include the overhead necessary to process private attributes internally. The additional steps required to support private attributes using the TE compliance checker necessitated that the verification and evaluation be accomplished externally to TE.

The TE policy compliance system is built to either use standard type comparisons defined directly in the policy or custom comparisons defined externally by a specified Java class. The ability to customize the comparison of credential content in an external class makes it possible to use the TE system for correctly verifying and comparing the private attributes of credentials. The decision to use custom external classes prevailed because TE had a history of success with TrustBuilder and there was no compliance checker similar to TE, but whose source could be easily modified. In order for an external class to be used during the compliance check, the class name and required certificate parameters must be specified in the TPL policy. A portion of an example TPL policy that defines a group and does not support private attributes is shown in Figure 13.

```
<GROUP NAME="SeniorStudent">
  <RULE>
    <INCLUSION FROM="University" ID="studentcert" TYPE="Student">
    </INCLUSION>
                    <FUNCTION>
                    <EQ>
                    <FIELD ID="studentcert" NAME="Status"></FIELD>
                            <CONST>Senior</CONST>
                    </EQ>
                    </FUNCTION>
  </RULE>
 </GROUP>
```

Figure 13 – Example TPL Policy without Private Attribute Support. The policy includes a constraint that the `status` field in the certificate must contain the constant value "Senior".

In contrast, a segment of a policy for the same scenario that does support external functions and uses an external class for a custom comparison is shown in Figure 14.

```
<GROUP NAME="SeniorStudent">
 <!--TP-COORD=(10,300)-->
 <RULE>
  <INCLUSION FROM="University" ID="studentcert" TYPE="Student"/>
      <FUNCTION>
      <EXTERN CLASS="PrivateAttributeExternalClass">
       <PARAM NAME="privateAttribute">
                <FIELD ID="studentcert" NAME="Status"/>
           </PARAM>
       <PARAM NAME="OID">
         <CONST>1.2.3.6.1</CONST>
           </PARAM>
       <PARAM NAME="valueEquals">
         <CONST>Senior</CONST>
           </PARAM>
       <PARAM NAME="subjectName">
         <FIELD ID="studentcert" NAME="subjectName"/>
           </PARAM>
       <PARAM NAME="issuerName">
         <FIELD ID="studentcert" NAME="issuerName"/>
           </PARAM>
      </EXTERN>
      </FUNCTION>
 </RULE>
</GROUP>
```

Figure 14 – Example TPL Policy with Private Attribute Support. The parameters specified are passed into the external class to be used to verify and evaluate private attributes.

In a nutshell, the TPL policy in Figure 14 states that in order to be a member of the `SeniorStudent` group, the private attribute found at the OID `1.2.3.4.6.1` of the certificate must evaluate to `Senior` in order to be valid. The policy refers to an external Java class named `PrivateAttributeExternalClass` and specifies five parameters to be sent into the external class through a series of method calls. The two main functions in the external class are `setParams()` and `eval()`. The TE compliance checker first calls `setParams()`, which initializes local variables within the external class to the respective parameters. The parameters are as follows: the private attribute digest, the OID, a custom comparison value, the certificate unique subject name and the certificate unique issuer name. Once each variable is set, the TE compliance checker calls `eval()` which evaluates the private attribute within the credential.

The `eval()` function takes three important steps. It obtains the sensitive pre-image from TrustBuilder, verifies the pre-image with its corresponding private attribute within the credential, and decides whether or not the sensitive information contained in the pre-image satisfies the policy.

The parameters specified by the policy provide identifying information to the external function regarding the certificate, the evaluation criteria, and the private attribute digest. The external function obtains the associated private attribute pre-images from TrustBuilder using a static method which takes unique identification information of the certificate in question, finds the correct negotiation session (in cases of multi-threaded operation) and returns the certificate extension's associated pre-image to the external function, if it exists. The code for accessing pre-images can be found in section 8.2.3 of the appendix.

Pre-image information can then be processed by the external class using the protocol described in section 2.4. This protocol describes the process of verification and evaluation that takes place in the `eval()` function. This function verifies a private attribute by hashing its private attribute pre-image and comparing the result to the private attribute itself. If the private attribute digests match, the original sensitive attribute value is extracted from the pre-image and evaluated to see whether the attribute value satisfies the trust requirements of the policy. A boolean value is returned to the compliance checker affirming whether or not the extracted value met the policy's requirements. The external function code can be found in section 8.2.2 of the appendix.

Because the compliance checker only determines if a set of credentials warrants membership in a group specified by the TPL policy, it is up to the negotiation strategy to determine if membership in a particular group merits access to protected resources such as credentials, services and pre-images.

4.4. A Negotiation Strategy for Selective Disclosure

The negotiation strategy determines the content of the messages sent to other TrustBuilder systems and controls the disclosure of credentials and resources. The "eager" strategy of trust negotiation can be summarized as negotiating trust by disclosing unprotected and available credentials at each pass of the negotiation without ever exchanging policies. Although both parties stay ignorant of the counterpart's trust requirements, if they present the right credentials to each other, they establish trust. The first working implementation of trust negotiation completed by the Internet Security Research Lab at BYU only supported the eager strategy. This eager strategy establishes trust with the fewest number of exchanges and may prove to be a very efficient strategy for establishing trust when supporting selective disclosure.

The eager strategy can be extended to support attribute level access control with private attributes by determining which private attribute pre-images to disclose as shown in Figure 15. The access control decision uses the compliance checker to determine which credentials gain membership of a group specified by the policy. The negotiation strategy handles the authorization and basically determines if membership to a particular group merits access to protected resources.

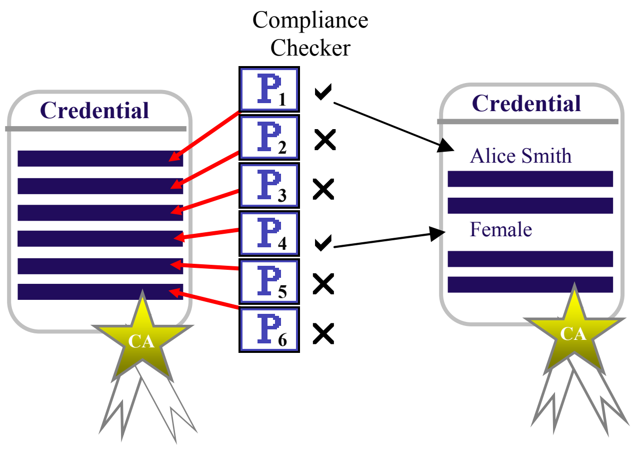


Figure 15 – Attribute Level Access Control. Attribute level access control is accomplished by associating policies with each sensitive attribute. During trust negotiation a sensitive value can be disclosed once its associated access control policy is satisfied.

In the previous section example, the TrustBuilder Agent's policy essentially states, "In order to receive the senior discount service at the bookstore, you must prove senior status at an accredited university." This statement of authorization is what we call a *role expression* and is explicitly defined in the TrustBuilder security agent's configuration file. This configuration file can express which resources are restricted, and to which groups they are restricted to. An example configuration file for TrustBuilder can be found in section 8.1.1 of the appendix. Written in XML format, the portion of the TrustBuilder security agent's configuration file that specifies a service access control policy is:

```
<SERVICE NAME="/senior_discount/books">
        POLICY>SeniorStudent</POLICY>
</SERVICE>.
```

Describing the access control for credentials in XML is very similar to the access control description for private attribute pre-images. The XML tags specific to pre-images include the name (Name) - denoting the attribute name, the source file (SRC) - for the pre-image file, and the extension identification (OID) - to uniquely identify the pre-image in the certificate. A sample portion of a TrustBuilder security agent's configuration file that specifies the disclosure policy for a private attribute pre-image is:

```
<CRED NAME="Student" SRC="demo/Certs/student.cer" SUPPORTING="false">
 <PREIMAGE Name="Status" SRC="demo/Preimages/student.txt"
OID="1.2.3.6.1">
  <POLICY>Company</POLICY>
 </PREIMAGE>
</CRED>
```

An approach for determining which credentials to disclose using the eager strategy is to iterate through the local credentials and check whether their associated disclosure policy is satisfied. If a credential's disclosure policy is satisfied, the credential is added to the vector of credentials to disclose. A check is then performed on the credentials to be disclosed and those credentials that had been previously disclosed, to determine which pre-images should be disclosed. The credentials and private attribute pre-images can be disclosed are placed into the negotiation response message, which is communicated to the other party. The complete negotiation strategy code is provided in section 8.2.5 of the appendix.

CHAPTER 5: RESULTS

This Chapter discusses how the size of credentials and the performance costs to process credentials is affected by private attributes. In addition, we also explore the impact private attributes have on trust negotiation.

5.1. Private Attribute Performance

Private attributes affect credentials by replacing extensions normally containing sensitive plain-text attribute values with private attribute digests. Each digest is currently a fixed length of 160 bits (20 bytes) corresponding to the output of the SHA-1 hash function. The binary output bytes are Base64 encoded to be represented as a printable string extending the 20 binary bytes into 28 ASCII bytes.

Private attributes affect the size of credentials and introduce an additional flow of private attribute pre-images which may altogether increase network traffic. Certificate extensions normally containing sensitive plain-text attribute values are replaced with private attribute digests. The certificate size may increase or even decrease depending on the original attribute's size. Each private attribute digest is a fixed length of 160 bits (20 bytes) corresponding to the output of the SHA-1 hash function. The binary output bytes are Base64 encoded to be represented as a printable string extending the 20 binary bytes into 28 ASCII bytes. Thus each attribute would be replaced by a 28 byte string.

Typically, X.509v3 certificates are nearly 1,000 bytes in size. As stated in section 3.2, private attributes may increase or decrease the size of the certificate based on the size of the data the private attribute digest represents. If the 28 byte private-attribute digest represents a sensitive 200,000 byte JPEG picture, the size of the credential would be decreased by 2 orders of magnitude, a substantial savings in network bandwidth and credential storage when a negotiating party does not require the JPEG pre-image associated with the certificate's private attribute. In general, the cost of storing credentials is only decreased when the sensitive data corresponding to the private-attribute digest is large and not necessary for every transaction. The cost for storing and distributing credentials is decreased or increased based on whether the sensitive attribute value is greater or less than the 28 byte digest.

The Java code for creating SHA-1 digests from pre-images is in section 8.2.1 of the appendix. The cost of creating private attribute digests is directly related to the cost of the one-way hash function used, which in this case is the cost of using the SHA-1 algorithm. The time it takes to create a 160-bit digest from a private attribute pre-image using SHA-1 on a Pentium 4, 1.8 GHz processor, with 512 Mb RAM averages around 10 milliseconds using a 131072 byte input as shown in Figure 16.

Assuming the average size of an attribute placed inside an extension of a certificate is less than 32 bytes, the creation and verification of that digest would cost less than .12 milliseconds on average to compute. Adding the hash function overhead of .12 milliseconds into a typical trust negotiation round time, the verification of pre-images becomes negligible as shown in Figure 17. If a private attribute pre-image is large in comparison to normal attributes, such as a JPEG picture file, latency will be introduced during pre-image message transmission and the verification of the corresponding private attribute digests within the certificates.

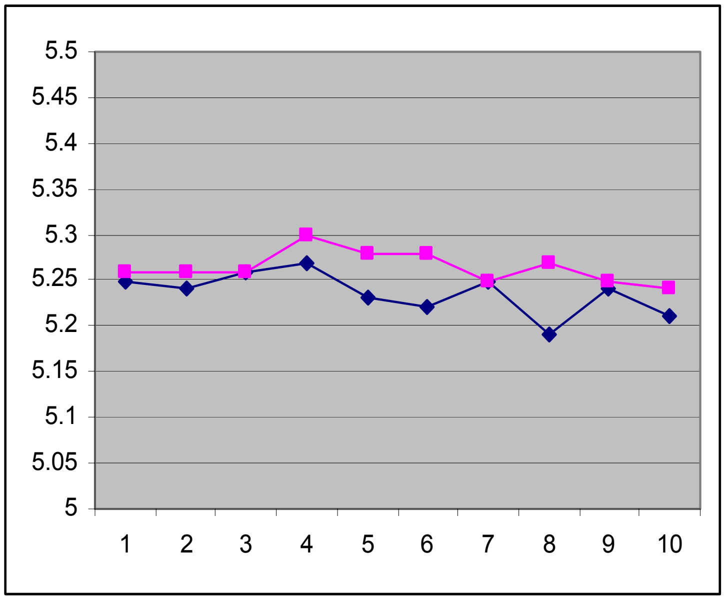
Private Attribute Verification Times		
Attribute Value Size (in bytes)	**Time to complete 1000 iterations of SHA-1 (in milliseconds)**	**Average time to create a single SHA-1 digest (in milliseconds)**
1	**120**	**0.12**
2	**120**	**0.12**
4	**120**	**0.12**
8	**120**	**0.12**
16	**120**	**0.12**
32	**120**	**0.12**
64	**121**	**0.121**
128	**133**	**0.133**
256	**140**	**0.14**
512	**165**	**0.165**
1024	**180**	**0.18**
2048	**241**	**0.241**
4096	**390**	**0.39**
8192	**671**	**0.671**
16384	**1201**	**1.201**
32768	**2323**	**2.323**
65536	**4627**	**4.627**
131072	**10355**	**10.355**

Figure 16 – Testing the Verification Time of Private Attribute Pre-images. The SHA-1 hashing method is used to create digests from pre-images to verify certificates during trust negotiation. The pre-image digest creation times are charted.

In Figure 17, we evaluate the cost of handling private attributes in trust negotiation, comparing it to a non-private attribute trust negotiation consisting of the same number of negotiating rounds. Each test measured the time it took from the HTTP request to the HTTP response using the HTTP implementation. On average, the cost of using private attributes was an additional 3 hundredths of a second, showing the time to complete a single trust negotiation with private attribute support to be negligible.

The Cost of Incorporating Private Attributes into Trust Negotiation

Run	Without Private Attributes	With Private Attributes
1	5.25	5.26
2	5.24	5.26
3	5.26	5.26
4	5.27	5.3
5	5.23	5.28
6	5.22	5.28
7	5.25	5.25
8	5.19	5.27
9	5.24	5.25
10	5.21	5.24
Max	5.27	5.3
Min	5.19	5.24
Average	5.236	5.265



*Time is measured in seconds —◆— Without Private Attributes —■— With Private Attributes

Figure 17 – The Cost of Private Attributes in Trust Negotiation. The graph and data show the contrast between identical scenarios involving the same number of rounds of trust negotiation; one test of trust negotiation over HTTP with private attribute support and the other without, showing the cost to be negligible.

5.2. The Impact of Private Attributes on Trust Negotiation

Supporting private attributes in trust negotiation furnishes credential holders with attribute level access control. This fine-grained access control helps to protect users from inadvertently disclosing their sensitive information. Selective disclosure of credential content using private attributes can help some trust negotiations succeed when previously they failed, and succeed faster in scenarios where the sensitive information is not necessary for the negotiation.

Suppose that Alice, an online shopper, has chosen several items to purchase and is ready to check out from Bob's online store. In order to establish the trust necessary for the transaction, Bob and Alice agree to negotiate trust. Bob requires that Alice send him a digital credential containing her name and email address before he will finish processing her online purchase request. Alice has a digital credential with her publicly available name and email address, but the credential also contains her social security number and weight, information that Alice considers sensitive. Without private attributes, all her attributes are stored in plain text within the credential and the disclosure policy is based on the sensitive information it contains, which in this case is the information not relevant to the transaction. Additionally, without private attributes, a credential's disclosure policy is logically composed of the conjunction of all its attributes' disclosure policies. If Bob cannot satisfy the trust requirements for the credential, the negotiation will fail. If Bob satisfies the trust requirements for the credential, he will receive Alice's weight and social security number, even though that information is not relevant to the transaction.

With attribute level access control, trust negotiation has the potential to increase its success rate and performance. This can be accomplished when sensitive attributes of a credential are hidden and the credential's disclosure policy is less restrictive, allowing it to be disclosed at an earlier stage of the negotiation. Using private attributes also allows Alice to hide the sensitive attributes within the credential and separate the policies governing the credential from the policies governing the credential's sensitive attributes. The credential's access control policy is less stringent, allowing Bob to access the credential and process the purchase request without inappropriately accessing Alice's sensitive information.

Selective disclosure of credential content empowers individuals to determine for themselves when, how and to what extent information about them is communicated to others during trust negotiation. Both the client and the server communicating in a trust negotiation have the capacity to protect and selectively reveal their sensitive information.

Trust negotiation is better equipped to succeed when possible using only the information that is relevant to the transaction. Thus the information that is collectively exchanged encompasses all the information required and nothing else. The usual irrelevant and extraneous information that typically escapes during transactions is not permitted to penetrate the fine-grained barrier of attribute level access control. This level of control is essential to protecting user privacy and discouraging the misuse and inappropriate access of private information.

CHAPTER 6: RELATED WORK

6.1. Crypto Certificates

Persiano et al. [13] describes *crypto certificates* as a way to disclose X.509v3 certificate extensions selectively by using public key cryptography to conceal the value stored in an extension. When the credential is issued, each sensitive attribute value is encrypted using the CA's public key, with the resulting cipher text stored in the certificate. No one can determine the original value from the cipher text unless they obtain the CA's private key. In order for a relying party to obtain reliably the sensitive attribute value, the certificate owner must disclose that value along with the certificate. The relying party can encrypt the value using the CA's public key and compare it to the cipher text stored in the certificate. If they match, the relying party accepts the attribute value disclosed by the owner as valid.

The proposal for crypto certificates does not mention anything about RSA padding. In order for crypto certificates to work as currently specified, no random padding can be added during RSA encryption. Otherwise, the relying party will not be able to generate the same cipher text for the attribute value that the CA generated. However, without random padding, the design is susceptible to dictionary attacks. Our approach to generating pre-images for private attributes could be adopted to prevent dictionary attacks against crypto certificates. In addition, the privacy features of private attributes are similar to crypto certificates, but at a reduced cost, since private attributes rely on one-way hashes instead of public key encryption.

6.2. Private Credentials

Brands [2] introduced *Private Credentials*, which are able to support anonymous, one-time-use credentials that cannot be traced to their owners. Private Credentials also support the selective disclosure of credential attributes and further support attribute disclosures that reveal certain properties of the attribute without revealing the actual value. For example, the credential owner could reveal that his or her age is over 18 without revealing an actual age.

Private Credentials are based on a new public key cryptography system that relies primarily on modular arithmetic, blinding factors, and mathematical proofs. The private key is a set of attributes $[a_1 \ldots a_n]$ and a chosen blinding factor s. The associated public key is the product of selected bases - a set of CA's public values $(b_1 \ldots b_k, h)$ - raised to the respective values of the private key $(b_1^{a_1} * \ldots * b_n^{a_n} * h^s)$. Because the public key mathematically includes the private key, the attributes in the private key are bound to the credential when the CA signs the public key. Credential holders can reveal their attributes selectively through proofs of knowledge using their private credential, the CA's signature, and public bases. This gives users fine-grained access control over the information in their Private Credentials. More technical details on Brand's private credentials can be found in [2]. Private Credentials are patented, and we are not aware of any publicly available implementation.

Like private attributes, Private Credentials can also be embedded as an extension in an X.509v3 certificate. Although this preserves the selective disclosure capabilities of Private Credentials, the anonymity and unlinkability properties inherent in Private Credentials are sacrificed because of the signatures contained in an X.509 certificate. In order to process Private Credentials, both parties in a transaction must support its public key cryptography system. In contrast, private attributes leverage existing hashing and public key cryptography standards, such as SHA-1 and RSA.

Private attributes support the selective disclosure of credential attributes. The design of private attributes presented in this thesis does not provide the sophisticated privacy properties of Private Credentials, including anonymity, untracability, and the ability to disclose certain information about an attribute value without disclosing the exact value. Private attributes are suitable for use in transactions that do not permit anonymity and the user trusts the server to handle the private information that is disclosed appropriately, such as in certain kinds of financial and medical transactions. In the future, we plan to experiment with Private Credentials in trust negotiation as soon as implementations become available.

6.3. Database Pointer

Renfro [15] proposes to protect the sensitive attributes of a certificate by placing the sensitive attributes in a centralized database and storing a database index for each attribute within the credential. The database's access control mechanism handles the selective disclosure of private information. Only those who are authorized by the database can gain access to the attributes. One advantage of this approach is being able to change the value of attributes in the database without re-issuing the credential. This would ease many burdens associated with managing information that may change frequently. In addition, to protect the integrity of an attribute, Renfro mentions that a hash of the attribute could be stored in the credential.

Using a database pointer approach, trust negotiation could be adopted to authenticate relying parties to the centralized database in open systems. The authorization required to access attributes from the database depends upon the database's access control mechanism, which will not give a user direct control over which or when sensitive information is disclosed.

6.4. Smart Certificates

Park and Sandhu [12] propose Smart Certificates and Secure Attribute Services on the web. They introduce the concept of an "attribute server" which is maintained by an attribute authority and issues attributes for users within the domain. Using this attribute server, users or servers can pull the appropriate attributes, verify them, and use them for their purposes, such as access control or authorization. The security of the attributes is dependent on the attribute server providing them securely to the Web server or user. In the server-pull architecture, the user would only have to provide authentication information and simply reference the attribute server where the user's attributes could be accessed. User-pull architecture allows the user to present her own attributes to whomever after downloading them from the attribute server. Smart Certificates allow sensitive information within the certificates to be encrypted, and then subsequently decrypted using an appropriate server private key or shared secret key. Smart Certificates and the Secure Attribute Services provide confidentiality of attributes as well as attribute level access control.

Smart Certificates allow symmetric or asymmetric encryption to provide confidentiality of sensitive attributes. Either way, the decryption using a server's private key or server's shared secret key would require that an intended recipient or server be explicitly known at the time the attribute is encrypted. In many applications, the intended recipients are not known prior to the transactions. In addition, any use of public key cryptography for encrypting and decryption attributes may carry significant performance costs.

Although it is possible to use Secure Attribute Services for selectively disclosing attribute information to others, if the server-pull architecture is used, the access control decision rests with the attribute server and not the owner of the attributes. The user-pull architecture is more preferable for privacy protection since the access control of attributes may be governed by the user, but currently would require user interaction unlike the automation of access control provided by TrustBuilder using private attributes.

CHAPTER 7: CONCLUSION AND FUTURE WORK

We presented an approach for protecting sensitive credential content during trust negotiation and demonstrated, through a design and an implementation, the privacy benefits that can be achieved with selective disclosure. Private attributes, created using a well-known technique incorporating bit commitments within digital credentials, are a simple and elegant way to achieve selective disclosure. This thesis is the first to explore this technique in detail and to provide a working implementation of selective disclosure in trust negotiation. Private attributes have the potential to impact all applications that process digital credentials because they can be easily integrated within standard X.509v3 certificates. A proof-of-concept implementation of private attributes was incorporated into TrustBuilder, an implementation of trust negotiation under development in the Internet Security Research Lab (ISRL) at BYU. The resulting implementation gives credential owners the ability to determine when and to whom sensitive information within their credentials is disclosed. This level of privacy control eliminates certain risks associated with exchanging credentials, including the excessive gathering of information that is not germane to the transaction, and inadvertently disclosing the value of a sensitive credential attribute. With access control at the attribute level, TrustBuilder gives users greater control over their private information and can improve the success rate of negotiations.

Future work includes implementing support for private attributes within TrustBuilder using a negotiation strategy that exchanges policies and incorporating Private Credentials into trust negotiation for enhanced privacy during transactions. We also intend to study the impact of private attributes on policy languages for trust negotiation, the impact of private attributes on trust negotiation protocols such as TNT [7], and incorporating private attributes within forthcoming XML credential formats. In addition, we will explore ways to extend the design of private attributes to include advanced privacy guarantees like anonymity, one-time-use credentials, and approximate query matching.

BIBLIOGRAPHY

[1] T. Barlow, A. Hess, and K. E. Seamons. Trust Negotiation in Electronic Markets. Eighth Research Symposium in Emerging Electronic Markets, Maastricht, Netherlands, September 2001

[2] S. A. Brands: Rethinking Public Key Infrastructures and Digital Certificates. MIT Press, Cambridge, Massachusetts, 2000.

[3] W. Diffie and M. Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, November 1976.

[4] C. M. Ellison. Establishing Identity without Certification Authorities. Proceedings of the Sixth Annual USENIX Security Symposium, pages 67–76, 1996.

[5] C. M. Ellison. Naming and Certificates. 10th Conference on Computers, Freedom, and Privacy, April 2000.

[6] A. Herzberg, J. Mihaeli, Y. Mass, D. Naor, and Y. Ravid. Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers. IEEE Symposium on Security and Privacy, Oakland, May 2000.

[7] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith. Advanced Client/Server Authentication in TLS. Network and Distributed System Security Symposium, San Diego, California, February 2002.

[8] R. Merkle. Protocols for Public Key Cryptosystems. In Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California, April 1980.

[9] R. Housley, W. Polk, W. Ford and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Request for Comments: 3280, Network Working Group, April 2002.

[10] M. Naor. Bit Commitment Using Pseudorandomness. Advances in Cryptology - Crypto 89. Lecture Notes in Computer Science, Vol. 435, Springer-Verlag, New York, 1990, pp. 128--137.

[11] National Institute of Standards and Technology. Secure Hash Standard. Federal Information, Processing Standards Publication 180-1, April 1995.

[12] J. S. Park and R. Sandhu. Smart Certificates: Extending X.509 for Secure Attribute Services on the Web. 22nd National Information Systems Security Conference, Crystal City, Virginia, October 1999.

[13] P. Persiano and I Visconti. User Privacy Issues Regarding Certificates and the TLS Protocol. 7th ACM Conference of Computer and Communications Security, Athens, Greece, November 2000.

[14] Recommendation X.509 Information Technology. Open Systems Interconnection. The Directory: Authentication Framework. International Telecommunication Union, August 1997.

[15] S. G. Renfro. VeriSign CZAG: Privacy Leak in X.509 Certificates. 11th USENIX Security Symposium, San Francisco, California, August 2002.

[16] R. Rivest, A. Shimir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Communications of the ACM, February, 1978.

[17] R. Rivest. The MD5 Message Digest Algorithm. Request for Comments: 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc. April 1992.

[18] B. Schneier. Applied Cryptography, 2nd edition. John Wiley & Sons, 1995.

[19] B. Schneier. Secrets and Lies, Digital Security in a Networked World. John Wiley & Sons, 1963.

[20] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobsen, H. Mills, and L. Yu. Requirements for Policy Languages for Trust Negotiation. 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), Monterey, California, June 2002.

[21] K. E. Seamons, M. Winslett, T. Yu, L. Yu, and R. Jarvis. Protecting Privacy during On-line Trust Negotiation. 2nd Workshop on Privacy Enhancing Technologies, San Francisco, California, April 2002.

[22] K. E. Seamons, M. Winslett, and T. Yu. Limiting the Disclosure of Access Control Policies during Automated Trust Negotiation. Network and Distributed System Security Symposium, San Diego, California, February 2001.

[23] K. E. Seamons, W. Winsborough, and M. Winslett. Internet Credential Acceptance Policies. Workshop on Logic Programming for Internet Applications, Leuven, Belgium, July 1997.

[24] W. Stallings. Network Security Essentials: Applications and Standards. Prentice Hall, New Jersey, 1999.

[25] A.F. Westin. Privacy and Freedom. Atheneum, New York, New York, 1967.

[26] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated Trust Negotiation. DARPA Information Survivability Conference and Exposition, Hilton Head, South Carolina, January 2000.

[27] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Negotiating Disclosure of Sensitive Credentials. Second Conference on Security in Communication Networks, Amalfi, Italy, September 1999.

[28] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. Negotiating Trust on the Web. IEEE Internet Computing, Volume 6, No. 6, November/December 2002.

[29] M. Winslett. An Introduction to Automated Trust Establishment. Workshop on Credential-Based Access Control, Dortmund, October 2002.

[30] T. Yu, M. Winslett, and K.E. Seamons. Automated Trust Negotiation over the Internet. 6th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Florida, July 2002.

[31] T. Yu, M. Winslett, and K. E. Seamons. Interoperable Strategies in Automated Trust Negotiation. 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, November 2001.

[32] T. Yu, M. Winslett, and K. E. Seamons. Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies for Automated Trust Negotiation. ACM Transactions on Information and System Security, volume 6, number 1, February 2003.

[33] T. Yu, X. Ma, and M. Winslett. PRUNES: An Efficient and Complete Strategy for Trust Negotiation over the Internet. ACM Conference on Computer and Communications Security, Athens, November 2000.

CHAPTER 8: APPENDIX

8.1. Scenario Data

8.1.1 TrustBuilder Configuration Files

SERVER XML FILE FOR TRUSTBUILDER

```
<CONFIG>
   <VAULT NAME="GENERICVAULT"/>
   <CREDENTIALS>
      <CRED NAME="JoeSmithSelfSigned" SELF="true"
      SRC="demo/Certs/JoeSmithSelfSigned.cer" SUPPORTING="true"/>
    <CRED NAME="University" SRC="demo/Certs/BYU.cer"
    SUPPORTING="false"/>
    <CRED NAME="Student" SRC="demo/Certs/me.cer" SUPPORTING="false">
      <PREIMAGE Name="Status" SRC="demo/Preimages/me.txt"
      OID="1.2.3.6.1">
            <POLICY>Company</POLICY>
      </PREIMAGE>
    </CRED>
    <CRED NAME="BBB" SRC="demo/Certs/BBB.cer" SUPPORTING="true"/>
   </CREDENTIALS>
   <DECISIONENGINE NAME="TE">
      <DECISIONENGINE_SUPPORT NAME="INI"
      SRC="demo/Config/Student_TPServer.ini"/>
      <DECISIONENGINE_SUPPORT NAME="POLICY" SRC="demo/Policy/student-
      policy.xml"/>
   </DECISIONENGINE>
   <OUTPUT NAME="File" CLASS="edu.byu.cs.isrl.output.FileOutput"/>
  <STRATEGIES>
      <FAMILY NAME="DTS" DEFAULT="EAGERSTRATEGY">
            <STRATEGY NAME="INFORMEDSTRATEGY"
            CLASS="edu.byu.cs.isrl.TrustBuilderHTTP.strategies.Informed
            "/>
         <STRATEGY NAME="EAGERSTRATEGY"
         CLASS="edu.byu.cs.isrl.TrustBuilderHTTP.strategies.Eager"/>
      </FAMILY>
   </STRATEGIES>
</CONFIG>
```

CLIENT XML FILE FOR TRUSTBUILDER

```
<CONFIG>
     <SERVICES>
         <SERVICE NAME="/order/books">
            <POLICY>Student</POLICY>
         </SERVICE>
      </SERVICES>
      <VAULT NAME="GENERICVAULT"/>
      <CREDENTIALS>
```

```
         <CRED NAME="AcmeBooksSelfSigned"
         SRC="demo/Certs/AcmeBooksSelfSigned.cer"
         SUPPORTING="true"/>
      <CRED NAME="AcmeBooks" SRC="demo/Certs/AcmeBooks.cer"
      SUPPORTING="false"/>
      <CRED NAME="ABET" SRC="demo/Certs/ABET.cer"
      SUPPORTING="true"/>
   </CREDENTIALS>
   <DECISIONENGINE NAME="TE">
         <DECISIONENGINE_SUPPORT NAME="INI"
         SRC="demo/Config/Store_TPServer.ini"/>
      <DECISIONENGINE_SUPPORT NAME="POLICY" SRC="demo/Policy/store-
      policy-custom-function.xml"/>
   </DECISIONENGINE>
   <OUTPUT NAME="File" CLASS="edu.byu.cs.isrl.output.FileOutput"/>
  <STRATEGIES>
      <FAMILY NAME="DTS" DEFAULT="EAGERSTRATEGY">
         <STRATEGY NAME="INFORMEDSTRATEGY"
         CLASS="edu.byu.cs.isrl.TrustBuilderHTTP.strategies.Informed
         "/>
         <STRATEGY NAME="EAGERSTRATEGY"
         CLASS="edu.byu.cs.isrl.TrustBuilderHTTP.strategies.Eager"/>
      </FAMILY>
   </STRATEGIES>
</CONFIG>
```

8.1.2 IBM TE Configuration Files

SERVER TE CONFIGURATION FILE

```
#TP Configuration
#Tue May 12 17:05:00 GMT+03:00 1998
Name=Acme Books
OrgUnit=Acme Retailers
Org=Acme Enterprises
Country=USA
Description=Acme Book Store
Port=2573
Policy=demo/Policy/store-policy-custom-function.xml
CertTypes=demo/Config/CertTypes.ini
Keys=demo/Vaults/store.key
DB=demo/Certs/CertDb_Store.txt
CrlDir=demo/Certs
VaultsDir=demo/Vaults
DescriptionUrl=http://isrl.cs.byu.edu
DebugLevel=1
Password=xxx
#
#To work with a flat file set Ldap=No and DB2=No
#
Ldap=No
DB2=No
#
DbHost=jdbc:db2:trust
#
DB2User=db2admin
```

```
DB2Password=webibm
#
CollectCerts=No
CollectCrls=No
```

CLIENT TE CONFIGURATION FILE

```
#TP Configuration
#Tue May 12 17:05:00 GMT+03:00 1998
Name=Joe Smith
OrgUnit=Student
Org=Brigham Young University
Country=USA
Description=BYU Student
Port=2574
Policy=demo/Policy/student-policy.xml
CertTypes=demo/Config/CertTypes.ini
Keys=demo/Vaults/student.key
DB=demo/Certs/CertDb_Student.txt
CrlDir=demo/Certs
VaultsDir=demo/Vaults
DescriptionUrl=http://isrl.cs.byu.edu
DebugLevel=1
Password=xxx
#
#To work with a flat file set Ldap=No and DB2=No
#
Ldap=No
DB2=No
#
DbHost=jdbc:db2:trust
#
DB2User=db2admin
DB2Password=webibm
#
CollectCerts=No
CollectCrls=No
```

8.1.3 IBM TPL Policies

SERVER TPL POLICY

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE POLICY SYSTEM "Policy.dtd">

<POLICY>
  <GROUP NAME="self">
    <!--TP-COORD=(10,30)-->
  </GROUP>
  <GROUP NAME="AccreditingBody">
    <!--TP-COORD=(10,100)-->
    <RULE>
         <INCLUSION FROM="self" ID="accreditingbodycert"
   TYPE="AccreditingBody">
        <!--TP-COORD=(0,0)-->
      </INCLUSION>
      </RULE>
    </GROUP>
      <GROUP NAME="University">
      <!--TP-COORD=(10,200)-->
      <RULE>
        <INCLUSION FROM="AccreditingBody" ID="universitycert"
     TYPE="University">
          <!--TP-COORD=(0,0)-->
        </INCLUSION>
      </RULE>
    </GROUP>
    <GROUP NAME="Student">
      <!--TP-COORD=(10,300)-->
      <RULE>
        <INCLUSION FROM="University" ID="studentcert" TYPE="Student"/>
     <FUNCTION>
     <EXTERN CLASS="PrivateAttributeExternalFunction">
                  <PARAM NAME="privateAttribute">
                 <FIELD ID="studentcert" NAME="Status"/>
           </PARAM>
                  <PARAM NAME="OID">
                          <CONST>1.2.3.6.1</CONST>
           </PARAM>
                  <PARAM NAME="subjectName">
                          <FIELD ID="studentcert" NAME="subjectName"/>
           </PARAM>
                  <PARAM NAME="issuerName">
                          <FIELD ID="studentcert" NAME="issuerName"/>
           </PARAM>
     </EXTERN>
     </FUNCTION>
      </RULE>
    </GROUP>
  </POLICY>\
```

CLIENT TPL POLICY

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE POLICY SYSTEM "Policy.dtd">

  <POLICY>
    <GROUP NAME="self"></GROUP>

    <GROUP NAME="AccreditingBody">
      <RULE>
      <INCLUSION FROM="self" ID="accreditingbodycert"
     TYPE="AccreditingBody"></INCLUSION>
      </RULE>
    </GROUP>

    <GROUP NAME="Company">
      <RULE>
     <INCLUSION FROM="AccreditingBody" ID="trustedbiz"
     TYPE="Store"></INCLUSION>
      </RULE>
  </GROUP>

</POLICY>
```

8.2. Source Code

8.2.1 Pre-image Creator Tool

```
/*
 * PrivateAttributeCreator.java
 *
 * Created on February 24, 2003, 1:51 PM
 */

/**
 *
 * @author  rjarvis
 */
import java.security.*;
import java.util.*;
import java.io.*;
public class PrivateAttributeCreator {

    public static void main(String[] args) {
    String Value = "";
        try{
        if(args[0] == null){
               System.out.println("Please specify Attribute Value
               file as an argument");
                System.exit(1);
        }
        if(args[1] == null){
        String PreimageFile = args[0].trim();
        Value = readPreimageFromFile(PreimageFile);
        }
        else{
        String TEST_FILE_LEN = args[1].trim();
        Value = "";
        //System.out.println("Integer Read in :" + TEST_FILE_LEN);
        int len = ((Integer) new Integer(TEST_FILE_LEN)).intValue();
        //System.out.println("Integer Read in :" + len);
            for(int x=0; x<(len/2); x++){
                Value += "AA";
            }
        }
         System.out.println("Length of Attribute Value in
         Characters: " + Value.length());
        //Create Random String
         Date randDate1 = new Date();
         SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
         random.setSeed((new Date().getTime()));
         byte bytes[] = new byte[20];
         random.nextBytes(bytes);
         String randomStr = new String(bytes);
         randomStr = new String(Base64.encode(bytes));
```

```
       // System.out.println("Size of randomStr = " +
       randomStr.length());
      //

      //Create the Preimage
       String preimage = Value + ":" + randomStr.substring(0, 20);
       Date randDate2 = new Date();
       System.out.println("Preimage created in Time:" +
       (randDate2.getTime()-randDate1.getTime()));
       //

      //Save the Preimage to File
      writeDataToFile("preimage.txt", preimage);

      //Create the Digest
      long time;
      Date randDate3 = new Date();
      String encodedDigest = "";
      for(int i=0; i<1000; i++){
          MessageDigest md = MessageDigest.getInstance("SHA");
          md.update(preimage.getBytes());
          byte[] digest = md.digest();
          encodedDigest = new String(Base64.encode(digest));
      }
      Date randDate4 = new Date();
      time = (randDate4.getTime() - randDate3.getTime());
       System.out.println("Preimage Digest created: " +
       encodedDigest + " in Time:" + time);

      //Save the Digest to File
       writeDataToFile("digest.txt", encodedDigest);
      }
      catch(Exception e){
          e.printStackTrace();
          System.exit(0);
      }
  }
 private static void writeDataToFile(String srcFile, String data){
  try{
          File file = new File(srcFile);
          BufferedWriter buff;
          file.delete();
          file.createNewFile();
          buff = new BufferedWriter(new FileWriter(file));
          buff.write(data);
          buff.close();
      }
      catch(Exception e){
          e.printStackTrace();
          System.exit(0);
      }
  }
  private static String readPreimageFromFile(String srcFile){
  String returnStr = "";
      try{
          File file = new File(srcFile);
          BufferedReader buff;
```

```
            if(!file.exists()){
                System.out.println("File NOT FOUND!");
                System.exit(1);
            }
            buff = new BufferedReader(new FileReader(file));
            returnStr = buff.readLine();
            buff.close();
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(0);
        }
    return returnStr;
    }
}
```

8.2.2 IBM TE External Function

```
/*
 * PrivateAttributeExternalFunction.java
 *
 * Created on February 5, 2003, 6:01 AM
 */
import edu.byu.cs.isrl.trustbuilder.TrustBuilder;
import com.ibm.trust.policy.*;
import java.security.*;
import java.util.*;
/**
 *
 * @author  rjarvis
 * @version 1.0
 */
public class PrivateAttributeExternalFunction implements
com.ibm.trust.policy.TPExternalFunction {

    public String subjectName;
    public String issuerName;
    public String OID;
    public String Value;
    public String privateAttribute;
    Date Timer;

    /** Creates new TPCompareTest */
    public PrivateAttributeExternalFunction() {
         System.out.println("Private Attribute External Function
         Evaluation...");
        subjectName = "";
        issuerName = "";
        OID = "";
        privateAttribute = "";

    }

   public int checkParam(java.lang.String str, java.lang.String
   str1) {
        return LEGAL_PARAM;
    }
```

```
public boolean eval() throws
com.ibm.trust.policy.TPExternalFunction$TPEvalException {

     boolean returnval = false;
      String preimage =
      TrustBuilder.sessionTable.getCertificatePreimage(subjectNam
      e, issuerName, OID);
     String attributeValue = extractValueFromPreimage(preimage);
     try {
         //Date t = new Date();
         MessageDigest md = MessageDigest.getInstance("SHA");
         md.update(preimage.getBytes());
         byte[] digest = md.digest();
         String output = new String(Base64.encode(digest));
         String output2 = privateAttribute;
         int result = output2.compareTo(output.trim());
         if (result == 0){
            System.out.println("**** The Private Attribute
            Verified****");
         }
         else{
            System.out.println("!!!! PRIVATE ATTRIBUTE NOT
            VERIFIED");
         }


         if(attributeValue.compareTo(Value) == 0)
            //The Attribute extracted from preimage should equal
            value passed in
             returnval = true;
         else returnval = false;

         if (returnval){
            System.out.println("**** The Private Attribute
            Satisfied Policy ****");
         }
         else{
            System.out.println("!!!! The Private Attribute DID
            NOT Satisfy Policy !!!!");
         }
     }
     catch (Exception ex) {
         ex.printStackTrace();
     }
     return returnval;
 }



public void setParam(java.lang.String str, java.lang.String str1)
{

     if (Character.toLowerCase(str.charAt(0)) == 's'){
         subjectName = str1;
         //System.out.println("subject");
     }else if (Character.toLowerCase(str.charAt(0)) == 'i'){
```

```
                issuerName = str1;
                //System.out.println("issuer");
            }else if (Character.toLowerCase(str.charAt(0)) == 'v'){
                Value = str1;
                //System.out.println("AttributeValue");
            }else if (Character.toLowerCase(str.charAt(0)) == 'o'){
                OID = str1;
                //System.out.println("OID");
            }else if (Character.toLowerCase(str.charAt(0)) == 'p'){
                privateAttribute = str1;
                //System.out.println("privateAttribute");
            }else{
                System.out.println("Not Recognized Parameter: " + str);
            }
        }
        private String extractValueFromPreimage(String preimage) {
            //Should have 20 bytes of random data
            String value = preimage.substring(0, preimage.indexOf(':'));
            return value;
        }

    }
```

8.2.3 TrustBuilder Internal Source Code

```
//PREIMAGE CLASS
package edu.byu.cs.isrl.trustbuilder;
import java.util.*;
import com.ibm.trust.certificate.*;
/*
 * Preimage.java
 *
 * Created on Feb 5, 2003, 9:41 AM
 */

/** This class is essentially a data store for preimages.
 * @author rjarvis
 * @version
 */
public class Preimage
{
    /** Holds value of property preimage. */
    private String preimage = null;
    /** Holds value of property credName. */
    private String credName = null;
    /** Holds value of property OID. */
    private String OID = null;
     /** Holds value of property preImageString. */
    private String preImageString = null;

    public Preimage()
    {

    }
    public Preimage(String preimageData){
       //Parse Preimage Data
       final String fNAME = "NAME:";
```

```
   final String fOID = "OID:";
   final String fDATA = "DATA:";
   int index1 = 0, index2 = 0;
   String temp1 = preimageData, temp2 = "";
   preImageString = preimageData;
   index1 = preimageData.indexOf(fNAME)+ fNAME.length();
   index2 = preimageData.indexOf(fOID);
   credName = preimageData.substring(index1,index2);
   index1 = index2 + fOID.length();
   index2 = preimageData.indexOf(fDATA);
   OID = preimageData.substring(index1, index2);
   index1 = index2 + fDATA.length();
   preimage = preimageData.substring(index1);

}
public String toString(){
  return "\tCredName: "+ credName + "\n\tOID: " + OID + "\n\tPre-
  image:" + preimage;
}
/** Getter for property preImageString.
* @return preImage.
 */

public String getPreImageString()
{
    return preImageString;
}

/** Setter for property preImageString.
 * @param cred New value of property preImageString.
 */
public void setPreImageString()
{
  this.preImageString = "NAME:" + credName + "OID:" + OID + "DATA:"
  + preimage;
}


/** Getter for property preImage.
 * @return preImage.
 */
public String getPreimage()
{
    return preimage;
}

/** Setter for property preimage.
 * @param cred New value of property preimage.
 */
public void setPreimage(String preimage)
{
    this.preimage = preimage;
}

/** Getter for property OID.
* @return OID.
*/
```

```
    public String getOID()
    {
        return OID;
    }

    /** Setter for property OID.
     * @param cred New value of property OID.
     */
    public void setOID(String OID)
    {
        this.OID = OID;
    }
      /** Getter for property credName.
     * @return OID.
     */
    public String getCredName()
    {
        return credName;
    }

    /** Setter for property credName.
     * @param cred New value of property credName.
     */
    public void setCredName(String credName)
    {
        this.credName = credName;
    }
}
```

STATIC METHOD FOR ACCESSING PREIMAGES

```
public String getCertificatePreimage(String subject, String issuer,
String OID){
        String preimage = "";
        Set keys = super.keySet();
        for(Iterator iter = keys.iterator(); iter.hasNext();)
        {
            String keyValue = (String)iter.next();
            Session s = (Session)super.get(keyValue);
            Iterator Creds = s.getRecvCredsIterator();
            while(Creds.hasNext()){
                TPCertificate cred = ((TPX509Certificate)Creds.next());
                String issuerStr = cred.getIssuer().getUniqueName();
                String subjectStr = cred.getSubject().getUniqueName();
                  if(issuerStr.compareTo(issuer) == 0 &&
                  subjectStr.compareTo(subject)== 0){
                    String hashedCredName =
                    TrustBuilder.doMD5Hash(cred.dumpToStringBase64NOLF(
                    ));
                    Iterator Preimages = s.getRecvPreimagesIterator();
                    while(Preimages.hasNext()){
                      Preimage preIMG = (Preimage)Preimages.next();
                        if(preIMG.getCredName().compareTo(hashedCredNam
                        e) == 0){
                            if(preIMG.getOID().compareTo(OID) == 0){
                              return preIMG.getPreimage();
                            }
                    }
                }
            }
        }
    }
    return preimage;
}
```

8.2.4 TrustBuilder Test Interfaces

```
/**
 * @author jarvis
 */
import java.io.*;
import com.ibm.trust.common.*;
import com.ibm.trust.TPServer.*;
import com.ibm.trust.certificate.*;
import com.ibm.trust.policy.*;
import java.util.*;
import java.net.*;
import edu.byu.cs.isrl.trustbuilder.*;
import edu.byu.cs.isrl.trustbuilder.util.*;

public class TrustTest_Client extends Thread implements TPCodes {
    public static void main(String[] args) {
        int result = 0;
        String pathToDemo = "demo/";
        String strFailed = "Failed to accept certificate ";
         String configFileURI = pathToDemo +
         "config/StudentConfig.xml";
        String preimage = "";
        Vector newCreds = new Vector();
        Vector newPreImages = new Vector();
        Vector newPolicies = new Vector();
        if(args[0] == null){
         System.out.println("Please specify argument s for soap and
         ns for so soap");
            System.exit(1);
        }
        try {
            if(Character.toLowerCase(args[0].charAt(0)) =='s'){
               TrustBuilderSOAP tb = new TrustBuilderSOAP(new
               URL("http://localhost:4041/soap/servlet/rpcrouter"));

            tb.init(configFileURI);
            SessionID id = tb.generateID();
            SessionID bogusId = tb.generateID();
            String AcmeCert = "-----BEGIN CERTIFICATE-----
MIICijCCAjSgAwIBAgIBDjANBgkqhkiG9w0BAQQFADBxMRMwEQYDVQQDEwpCQkIgT25saW5
lMR8wHQYDVQQLExZCZXR0ZXIgQnVzaW5lc3MgQnVyZWF1MSswKQYDVQQKEyJDb3VuY2lsIG
9mIEJldHRlciBCdXNpbmVzcyBCdXJlYXVzMQwwCgYDVQQGEwNVU0EwHhcNMDIwODIzMTcyM
DEzWhcNMDMwODIzMTcyMDEzWjBXMRMwEQYDVQQDEwpBY21lIEJvb2tzMRcwFQYDVQQLEw5B
Y21lIFJldGFpbGVyczEZMBcGA1UEChMQQWNtZSBFbnRlcnByaXNlczEMMAoGA1UEBhMDVVN
BMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJAjzKr5b3s5Y7KqLRZzne30HsUbUmRaEv42td
cHWgDn/clacEzephtqZG54yyaz55WnlfqNMQgJMat+S+nDMyMCAwEAAaOB0DCBzTAjBgNVH
RIEHDAaghhJb3U4Umt5MzlRRW9sYjhYWkZQd2J3PT0wIwYDVR0RBBwwGoIYOWpjVGRtSlo2
```

```
WjdWVXpad2huaG5tQT09MA0GBCoDBAEEBVN0b3JlMBAGBCoDBAIECFZlcnNpb24xMB4GBCo
DBAMEFmh0dHA6Ly9pc3JsLmNzLmJ5dS5lZHUwEwYEKgMEBAQLV0lDS0VUOjI1NzMwEwYEKg
MEBQQLV0lDS0VUOjI1NzMwFgYEKgMGAQQOQWNtZSBCb29rc3RvcmUwDQYJKoZIhvcNAQEEB
QADQQBNgc3SjncUibqX1VkktAYJdlMtgeSE52Z1W0c8uK9kKe4NWECQlp7OCWHHTeED6B/7
LEGe6V/v8AH3Ucsm9mr1-----END CERTIFICATE-----";

            System.out.println("\n ***** ROUND ONE ****** \n");
            System.out.println("Joe requests the bookstore student
            discount from AcmeBooks.com, an online bookstore... ");
            System.out.println("\n ***** ROUND TWO ****** \n");
            newCreds.removeAllElements();
            System.out.println("Server negotiates using the Eager
            Strategy \n sends his publically available
            Certificates/Preimages");
            newCreds.add(AcmeCert);
            NegotiationResponse nr = tb.negotiate(newCreds,
            newPolicies, newPreImages, id);
            System.out.println("\n>>>>>>>>>>>\tJoe Sends to Server:
            \n>>>>>>>>>>>\t");
            System.out.println("Number of Preimages: " +
            nr.getPreImagesToDisclose().size());
            printCredVector(nr.getCredsToDisclose());
            printPreimageVector(nr.getPreImagesToDisclose());
            if(!nr.isAuthenticated()){
                  System.out.println("\nTrust requirements not met,
                  continuing Trust Negotiation...");
               }
               else{
                  System.out.println("\n\tTrust requirements
                  met!\n\tTrust Negotiation Succeeded! \n");
               }
          }else{
            TrustBuilder tb = new TrustBuilder();
            tb.init(configFileURI);
            SessionID id = tb.generateID();
            SessionID bogusId = tb.generateID();
            String AcmeCert = "-----BEGIN CERTIFICATE-----
MIICijCCAjSgAwIBAgIBDjANBgkqhkiG9w0BAQQFADBxMRMwEQYDVQQDEwpCQkIgT25saW5
lMR8wHQYDVQQLExZCZXR0ZXIgQnVzaW5lc3MgQnVyZWF1MSswKQYDVQQKEyJDb3VuY2lsIG
9mIEJldHRlciBCdXNpbmVzcyBCdXJlYXVzMQwwCgYDVQQGEwNVU0EwHhcNMDIwODIzMTcyM
DEzWhcNMDMwODIzMTcyMDEzWjBXMRMwEQYDVQQDEwpBY21lIEJvb2tzMRcwFQYDVQQLEw5B
Y21lIFJldGFpbGVyczEZMBcGA1UEChMQQWNtZSBFbnRlcnByaXNlczEMMAoGA1UEBhMDVVN
BMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAJAjzKr5b3s5Y7KqLRZzne30HsUbUmRaEv42td
cHWgDn/clacEzephtqZG54yyaz55WnlfqNMQgJMat+S+nDMyMCAwEAAaOB0DCBzTAjBgNVH
RIEHDAaghhJb3U4Umt5MzlRRW9sYjhYWkZQd2J3PT0wIwYDVR0RBBwwGoIYOWpjVGRtSlo2
WjdWVXpad2huaG5tQT09MA0GBCoDBAEEBVN0b3JlMBAGBCoDBAIECFZlcnNpb24xMB4GBCo
DBAMEFmh0dHA6Ly9pc3JsLmNzLmJ5dS5lZHUwEwYEKgMEBAQLV0lDS0VUOjI1NzMwEwYEKg
MEBQQLV0lDS0VUOjI1NzMwFgYEKgMGAQQOQWNtZSBCb29rc3RvcmUwDQYJKoZIhvcNAQEEB
QADQQBNgc3SjncUibqX1VkktAYJdlMtgeSE52Z1W0c8uK9kKe4NWECQlp7OCWHHTeED6B/7
LEGe6V/v8AH3Ucsm9mr1-----END CERTIFICATE-----";
            System.out.println("\n ***** ROUND ONE ****** \n");
            System.out.println("Joe requests the bookstore student
            discount from AcmeBooks.com, an online bookstore... ");
            System.out.println("\n ***** ROUND TWO ****** \n");
            newCreds.removeAllElements();
            System.out.println("Server negotiates using the Eager
            Strategy \n sends his publically available
            Certificates/Preimages");
            newCreds.add(AcmeCert);
            NegotiationResponse nr = tb.negotiate(newCreds,
            newPolicies, newPreImages, id);
            System.out.println("\n>>>>>>>>>>>>\tJoe Sends to Server:
            \n>>>>>>>>>>>>\t");
            System.out.println("Number of Preimages: " +
            nr.getPreImagesToDisclose().size());
            printCredVector(nr.getCredsToDisclose());
            printPreimageVector(nr.getPreImagesToDisclose());
            if(!nr.isAuthenticated()){
                  System.out.println("\nTrust requirements not met,
                  continuing Trust Negotiation...");
               }
               else{
                  System.out.println("\n\tTrust requirements
                  met!\n\tTrust Negotiation Succeeded! \n");
               }

      }
      catch (Exception e) {
          e.printStackTrace();
          System.exit(1);
      }
       }
       public static void printCredVector(Vector Vstring){
      try{
           System.out.println("\n\tCredentials Received:");
      for(int i=0; i < Vstring.size(); i++){
               //TPCertificate myTempCert =
            TPX509Certificate.loadCertFromString((String)Vstring.elemen
            tAt(i));
            System.out.println(Vstring.elementAt(i));
      }
           }catch(Exception e){
               e.printStackTrace();
           }
       }
       public static void printPreimageVector(Vector Vstring){
      try{
      System.out.println("\n\tPREIMAGES Received:");
      for(int i=0; i < Vstring.size(); i++){
         System.out.println(Vstring.elementAt(i).toString());
      }
           }catch(Exception e){
               e.printStackTrace();
           }
       }
   }
```

8.2.5 TrustBuilder Eager Strategy Code

```
   /*
    * EagerStrategy.java
    *
```

```
 * Created on May 16, 2002, 1:45 PM
 */

package edu.byu.cs.isrl.trustbuilder;
import java.util.*;
import com.ibm.trust.certificate.*;
/**
 *
 * @author bsmith, rjarvis
 */
public class EagerStrategy implements Strategy {

    Vault vault = null;

    /** Creates a new instance of EagerStrategy */
    public EagerStrategy(Vault v) {
        this.vault = v;
    }
   public NegotiationResponse runStrategy(Session sessionData,
   AccessControl accessControl, DecisionEngine decisionEngine)
    {
   return runStrategyMethod(sessionData, accessControl,
   decisionEngine, null);
    }
   public NegotiationResponse runStrategy(Session sessionData,
   AccessControl accessControl, DecisionEngine decisionEngine,String
   resource)
    {
   return runStrategyMethod(sessionData, accessControl,
   decisionEngine, resource);
    }
    /** places all unlocked credentials in the Negotiation class
     */
   private NegotiationResponse runStrategyMethod(Session
   sessionData, AccessControl accessControl, DecisionEngine
   decisionEngine, String resource){
    NegotiationResponse nr = new NegotiationResponse();
    Vector aRoles = new Vector(sessionData.getAuthenticatedRoles());
    Vector OIDs = new Vector();
    if(resource != null){
      // check if resource is protected
   RoleExpression resourceExpr = (RoleExpression)
   accessControl.get(resource);
      //System.out.println(resourceExpr);
      if (resourceExpr.Evaluate(aRoles))
         nr.setAuthenticated(true);
    }
    // find unlocked credentials
    Enumeration keys = accessControl.keys();
    while (keys.hasMoreElements()) {
        Object key = keys.nextElement();

        //System.out.println("Key:\t" + (String)key);
         RoleExpression credExpr = (RoleExpression)
         accessControl.get(key);
        //System.out.println(credExpr);
        if (credExpr.Evaluate(aRoles)) {
            if(isOID((String)key)){
                OIDs.add((String)key);
                //System.out.println("OID ADDED: " + (String)key);
                sessionData.addToUnlockedPreimages((String)key);
            }
            else{
                TPX509Certificate cred =
               (TPX509Certificate)vault.getCred((String)key);
                if(cred == null) continue;
             nr.addToCredsToDisclose(cred.dumpToStringBase64NOLF());
                sessionData.addToUnlockedCreds(key);
                sessionData.addToDisclosedCreds(key);
            }
        }
    }
    Iterator UnlockedCreds = sessionData.getUnlockedCredsIterator();
    while(UnlockedCreds.hasNext()){
        String CredName = (String)UnlockedCreds.next();
        for(int i=0; i < OIDs.size(); i++){
          String PreimageStr = vault.getPreimage(CredName,
         ((String)OIDs.elementAt(i)));
            if(PreimageStr != null){
            nr.addToPreImagesToDisclose(PreimageStr);
            sessionData.addToDisclosedPreimages(PreimageStr);
            }
        }
    }
   // System.out.println("In Eager Strategy!");
    return nr;
}
public boolean isOID(String s){
    //Verify that the string contains number.number format
    boolean firstCharISADigit = Character.isDigit(s.charAt(0));
    int index = s.indexOf(".");
    if(firstCharISADigit && index > 0)
        return true;
    return false;
}
}
```