# TRUST NEGOTIATION IN SESSION-LAYER PROTOCOLS

by

Jared Jacobson

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

July 2003

Copyright © 2003 Jared Jacobson

Permission is granted to use this work or any portion of it for any purpose as long as sufficient context is given around the citation to correctly explain the author's original intent.

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Jared Jacobson

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

| | |
|---|---|
| Date | Kent E. Seamons, Chair |
| Date | Mark J. Clement |
| Date | Robert P. Burton |

BRIGHAM YOUNG UNIVERSITY

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

| Date | Kent E. Seamons<br>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

TRUST NEGOTIATION IN SESSION-LAYER PROTOCOLS

Jared Jacobson

Department of Computer Science

Master of Science

Identity-based authentication is insufficient for many real-world interactions. Real-world interactions may include access requests for a number of secure resources or services over the course of a single session. Automated trust negotiation addresses the problem of authenticating entities with no pre-existing relationship. However, trust negotiation as it is presently understood has no concept of a session. This thesis considers how to incorporate trust negotiation into existing secure session-layer network protocols and discusses a few simple ways to make authentication over the course of a session more efficient by caching authentication information.

ACKNOWLEDGMENTS

Thanks go to my wife Kim, who lost me to this thesis for a few months surrounding the birth of our first child. Thanks are further due to Dr. Kent Seamons, who developed the idea of trust negotiation in the first place, and has been a kind and understanding teacher, mentor, and friend. Thanks are also due to Hyrum Mills, who helped me with the implementation of TNT, and Jim Henshaw, who helped with the implementation of TNE-SSH.

## Contents


**1 Introduction** **1**

1.1 The Problem . . . . . 1

1.2 Trust Negotiation . . . . . 2

1.2.1 Sensitive Credentials . . . . . 2

1.2.2 Sensitive Policies . . . . . 3

1.2.3 Negotiation Strategies . . . . . 4

1.3 Trust Negotiation Examples . . . . . 4

1.3.1 Pharmacy Example . . . . . 5

1.3.2 Open-Source Development . . . . . 5

1.3.3 Credit Card Example . . . . . 6

1.4 Motivation for the Thesis . . . . . 6

1.5 Thesis Statement . . . . . 9

1.6 Summary of Upcoming Chapters . . . . . 9

**2 Background Material** **11**

2.1 Symmetric-Key Cryptography . . . . . 11

2.2 Public-Key Cryptography . . . . . 12

2.3 Digital Credentials . . . . . 15

2.4 Credential Chaining . . . . . 17

2.5 Access-Control Policies . . . . . 18

**3 Related Work** **19**

3.1 Session-layer Protocols . . . . . 19

3.2 Identity-based Access Control . . . . . 19

3.3 Role-based Access Control . . . . . . . . . . . . . . . . . . . . . . . . 20
3.4 Trust Management . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.5 Trust Negotiation . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**4 Session-Layer Trust Negotiation 23**

4.1 Session-Layer Trust Negotiation . . . . . . . . . . . . . . . . . . . . 23
4.2 Trust Caching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.3 Trust Accumulation . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.4 Session-Layer Protocols . . . . . . . . . . . . . . . . . . . . . . . . 25

**5 TLS Analysis 27**

5.1 TLS Authentication . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.2 Limitations of TLS Authentication . . . . . . . . . . . . . . . . . . 30
5.3 Extending TLS to Support Trust Negotiation . . . . . . . . . . . . . 31
5.3.1 TLS Rehandshake . . . . . . . . . . . . . . . . . . . . . . . . 31
5.3.2 TLS Session Resumption . . . . . . . . . . . . . . . . . . . . 32
5.3.3 TLS Message Extensions . . . . . . . . . . . . . . . . . . . . 33
5.3.4 Facilitating Trust Negotiation in TLS . . . . . . . . . . . . . . 33
5.3.5 Negotiation Failure in TLS . . . . . . . . . . . . . . . . . . . 37
5.4 Backward Compatibility . . . . . . . . . . . . . . . . . . . . . . . . 38
5.5 Caching and Accumulation . . . . . . . . . . . . . . . . . . . . . . . 39
5.6 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**6 SSH Analysis 43**

6.1 SSH Authentication . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.2 SSH User Authentication . . . . . . . . . . . . . . . . . . . . . . . 45
6.3 Limitations of SSH Authentication . . . . . . . . . . . . . . . . . . 47

6.4 Extending SSH to Support Trust Negotiation . . . . . . . . . . . . . 48
6.4.1 Requesting Trust Negotiation as the First Service . . . . . . . 50
6.4.2 Facilitating Trust Negotiation in SSH . . . . . . . . . . . . . . 51
6.5 Backward Compatibility . . . . . . . . . . . . . . . . . . . . . . . . 54
6.6 Caching and Accumulation . . . . . . . . . . . . . . . . . . . . . . 55
6.7 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**7 Implementation Notes** **57**
7.1 TrustBuilder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.2 TLS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.2.1 TNT Negotiation Overview . . . . . . . . . . . . . . . . . . . 57
7.2.2 TLS Data Types . . . . . . . . . . . . . . . . . . . . . . . . . 58
7.2.3 TNT Message Definitions . . . . . . . . . . . . . . . . . . . . 61
7.3 SSH . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
7.3.1 SSH Data Types . . . . . . . . . . . . . . . . . . . . . . . . . 64
7.3.2 TNE-SSH Messages . . . . . . . . . . . . . . . . . . . . . . . . 65
7.3.3 TNE-SSH Implementation Considerations . . . . . . . . . . . 71

**8 Conclusion and Future Work** **73**
8.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
8.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

**A SSH Constants** **77**

## List of Tables


5.1 TLS/TNT Compatibility . . . . . . . . . . . . . . . . . . . . . . . . 38

6.1 SSH Protocol Layers . . . . . . . . . . . . . . . . . . . . . . . . . 43

7.1 SSH Data Types . . . . . . . . . . . . . . . . . . . . . . . . . . 65

A.1 Partitioning of the Byte Value Message Identifier for SSH . . . . . . . 77

A.2 Constant Values for SSH Messages . . . . . . . . . . . . . . . . . . 78

A.3 Constant Values for Negotiation Failure in SSH . . . . . . . . . . . . 78

## List of Figures


1.1 Authentication styles . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Trust negotiation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

5.1 TLS messaging . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

5.2 TNT messaging . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

6.1 SSH user authentication . . . . . . . . . . . . . . . . . . . . . . . 46

6.2 SSH with trust negotiation . . . . . . . . . . . . . . . . . . . . . . 50

6.3 Trust negotiation in SSH . . . . . . . . . . . . . . . . . . . . . . . 51

7.1 TNT architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

# Chapter 1 — Introduction

## 1.1 The Problem

Identity-based authentication is insufficient for many real-world interactions. Real-world interactions may include access requests for a number of secure resources or services over the course of a single session. Automated trust negotiation [19, 20, 26] addresses the problem of authenticating entities with no pre-existing relationship. However, trust negotiation as it is presently understood has no concept of a session. This thesis considers how to incorporate trust negotiation into existing secure session-layer network protocols and discusses a few simple ways to make authentication over the course of a session more efficient by caching authentication information.

The idea of a session is common in networked computing. A session consists of a number of messages exchanged between two entities over the course of a single interaction. For example, an SSH session begins with a user authenticating to the server. After authenticating, the client can then execute any commands that he could execute if he were logged onto the system locally.

Growing concern for security in networked environments has prompted the development of a large number of authentication methods by which a server can determine whether a user should be allowed access to a service or services. Most of these authentication methods do so based solely on the user's identity. Identity-based authentication is appropriate when the client and server applications share a security domain, but when they don't, authentication becomes awkward.

Current research is developing an approach for authenticating strangers known as *automated trust negotiation* (described in the next section). For use in session-layer network protocols (such as FTP, SSH, or TLS), trust negotiation as presently

designed is inefficient when a session includes multiple requests for secure resources. This thesis describes two ways to improve the efficiency of authentication using trust negotiation in session-layer protocols: trust caching and accumulation.

## 1.2 Trust Negotiation

Automated trust negotiation addresses the problem of authenticating strangers using digital credentials that contain properties (referred to as attributes) about the user who is trying to gain access to a particular service. Such credentials are cryptographically signed by a trusted third party who verifies that the attributes contained in the credential are accurate and correct. So when a service owner writes the access-control policy for his service, he can specify what attributes a client must have to gain access to the service. By submitting credentials that contain the attributes specified in the access-control policy, a client gains access to the service. A server or client specifies who they will trust to assert certain attributes about an individual by maintaining a list of trusted third parties for a particular attribute.

Traditional access-control methods map a user's identity to permissions (Fig. 1.1a). More recently, research has developed another method of access control, referred to as role-based access control (RBAC)[5]. In this scheme, users are mapped to roles, which are then mapped to permissions (Fig. 1.1b). This simplifies security administration and allows for more flexible access-control policies. It also allows deployment of the principle of least privilege: no more privilege is used to execute a command or access a file than is necessary. Trust negotiation leverages RBAC by mapping user attributes to roles, which are then mapped to permissions (Fig. 1.1c).

### 1.2.1 Sensitive Credentials

Credentials may contain sensitive information, which includes anything that should not be naïvely disclosed to strangers: Social Security Number, driver license number, credit card number, weight, and so forth. Sensitive credentials are guarded by

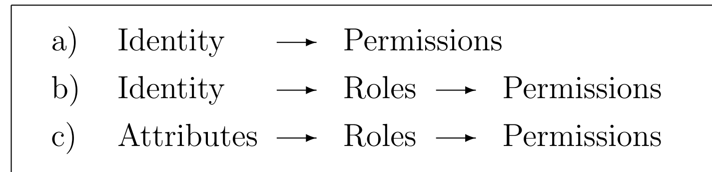


Figure 1.1: Authentication styles: a) Standard username/password style b) Role-based c) Trust negotiation

access-control policies (or simply *policies*) that enumerate the attributes that the other negotiator must satisfy to receive the credential. Credentials that are guarded by a policy that the other negotiator has not yet met are referred to as *locked*. After the other negotiator presents evidence that he meets the access-control policy, the credential is *unlocked*. Finally, when the other negotiator is sent the unlocked credential, the credential is considered *released*.

### 1.2.2 Sensitive Policies

To expedite the process of negotiating trust, it is desirable that each negotiator know what the other negotiator's requirements are for gaining access to a particular credential or service. To this end, it is recommended that negotiators exchange access-control policy information [19, 16]. However, policies can also contain sensitive information, as illustrated in the following example.

#### 1.2.2.1 Secret Collaboration Example

Suppose that Monopolysoft and SecurityLeak, two software companies, are secretly cooperating to develop a user-friendly desktop firewall so powerful and so secure that it will sink the companies that have popular firewalls on the market now. To simplify administration of their inter-corporate project, each company gives its employees who work on the project a credential containing the attribute "fireman". Every project resource is given an access-control policy that states that only people who have credentials containing a "fireman" attribute that are issued by Monopolysoft

or SecurityLeak can have access.

While this policy prevents unauthorized access to the firewall project, it does not protect another sensitive bit of information: the existence of the secret collaboration between Monopolysoft and SecurityLeak. Because the collaboration is secret, it is vital that this policy not get into the hands of the competing firewall companies. Therefore the policy must be protected under another access-control policy.

### 1.2.3 Negotiation Strategies

Since both the client and server may have sensitive credentials and policies, trust negotiation may require several exchanges of credentials and requests for credentials—hence the term trust *negotiation* (see Fig. 1.2). Each negotiator follows a trust negotiation *strategy*, which defines when in the negotiation his unlocked credentials can be released. Since there are a potentially infinite variety of negotiation strategies, strategies are grouped into *strategy families*. All of the strategies within a given strategy family are guaranteed to be mutually interoperable—that is, they are guaranteed to succeed whenever success is possible. Strategies from different families are not guaranteed to interoperate.

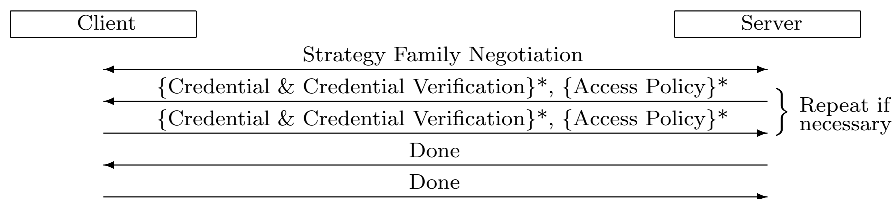


Figure 1.2: Trust negotiation

## 1.3 Trust Negotiation Examples

This section contains a number of example scenarios that illustrate situations in which trust negotiation is useful. Trust negotiation in TLS would enable the pharmacy and credit-card examples. Adding trust negotiation to SSH would allow

the open-source development example.

### 1.3.1 Pharmacy Example

Suppose that Alice has a pharmaceuticals company and wants to run a service whereby state-licensed pharmacies can order medicines online. Using identity-based authentication, Alice would have to maintain a list of every licensed pharmacy so that when a pharmacy makes a purchase request she can verify that the visitor is, in fact, a licensed pharmacy. Even then, if she receives a request in the name of a licensed pharmacy, how does Alice know that the accessor is from that pharmacy? Having no pre-existent relationship with the pharmacy, it would be presumptuous of Alice to issue a password to the accessor without some kind of verification. Clearly, identity-based authentication is not an acceptable solution.

Suppose that each state's board of pharmacy issues digital credentials to the state's licensed pharmacies containing attributes relevant to the pharmacy (including, for example, its location and whether or not it is allowed to distribute radiopharmaceuticals). When Charlie opens a new general pharmacy in the state of Massachusetts, he obtains a license from the board of registration in pharmacy. Charlie's pharmacy credential will not contain the attribute of a radiopharmaceuticals distributor (since by Massachusetts law nuclear pharmacies may only distribute nuclear medicines). Alice can then create the policy that only those licensed pharmacies that contain the appropriate attribute can order her company's radiopharmaceuticals, but any licensed, non-nuclear pharmacy can order her company's normal pharmaceuticals.

### 1.3.2 Open-Source Development

Having heard from an anonymous source that Monopolysoft and SecurityLeak are creating a firewall, an open-source development team decides to create a firewall to compete with it. The team has set up CVS access to the program source. The CVS is set up for anonymous read access, so that anyone may download the source and

modify it. However, in order to maintain rigorous coding standards on the project, the team restricts access write access to the source files: Only a Red Hat Certified Developer or Microsoft Certified Solution Developer may commit modifications to the files. For certain core files, modifications must be cleared through the project administrator before they can be committed—the project administrator gives the developer a single-use commit credential when his modifications are accepted.

### 1.3.3 Credit Card Example

Fifth Federal Bank will give minors a credit card if their parents co-sign on the account. Fifth Federal requires that a parent (or legal guardian) create the account. The parent must have a valid credit card and a good credit history to create the account. The bank obtains a credit report automatically when the request is made. (Fifth Federal takes the appropriate steps to inform the parent or guardian that this will occur.) Further, the parent must present the child's digital birth certificate or court-signed guardianship credential (that names the parent as guardian), in order to validate the transaction. Fifth Federal has a federal charter credential that specifies it as a bank.

Mary wants to give her son, Michael, a credit card for his birthday. She has the credentials that Fifth Federal requires. She has the policy that she only gives her credit card credential to federal banks, credit unions, and state-licensed companies. She will give Michael's birth certificate credential to anyone who requests it.

## 1.4 Motivation for the Thesis

Trust negotiation presently performs per-request access-control. That is, for each sensitive service or resource requested, trust negotiation occurs. Even if the same resource is requested multiple times, a trust negotiation occurs for each request. This is obviously wasteful. If the resource requester has already authenticated sufficiently to receive a resource during this session, why should he have to reauthenticate for the

same resource?

In an existing implementation of trust negotiation, the pharmacy example (which is quite simple, as trust negotiations go) takes approximately three seconds and involves the transfer of more than 6 KB of negotiation information. More complex negotiations can take substantially longer. Suppose that during an SSH session, the client executes a command that accesses five resources that all have the same policy. If the negotiation is as simple as the pharmacy example, the negotiation will add fifteen seconds of overhead to the command execution. If the server caches the authentication information already established, the negotiation will only add three seconds of overhead—and the client and server will never have to negotiate to determine whether the client fulfills the policy for the remainder of the session.

To illustrate the potential consequences of negotiating on every access request, suppose that the command is executed in a shell script 800 times. Running the script will take 120,000 seconds of negotiation time on a server that does not cache the client's authentication information. It will also result in the transfer of almost 23 1/2 MB of data over the network. Ignoring IP, TCP, and session-layer protocol packet headers, that would take more than 3500 seconds to transfer over a modem, adding almost an hour of network latency to running the script. That's a little over 34 hours, total. If the script runs in two seconds for someone logged in locally to the server, uncached trust negotiation introduces a 61,750 percent increase in the time taken to run the script. A server that caches the authentication information will negotiate once with the client, and upon successful completion of the negotiation, will not need to negotiate again. This introduces a total of three seconds of overhead to running the script the first time—a 150 percent increase. If the script is run again during the session, the client and server will not have to negotiate at all.

Caching authentication information allows the accumulation of authenticated roles,

so that determining whether or not a user can have access to a resource gets more and more efficient over the course of a session: those roles that have already been negotiated once do not need to be negotiated again. This thesis demonstrates that trust negotiation can be incorporated into session-layer security protocols without breaking the existing protocols, and discusses the implications of caching the trust accumulated during a session.

Enabling existing session-layer protocols to perform trust negotiation and cache authentication information has a few advantages over requiring applications to negotiate trust themselves.

Secure code is difficult to write, and it is beneficial to reuse a well-tested, heavily reviewed code base rather than having each application rewrite the same protocols and potentially introduce subtle insecurities. Pushing trust negotiation down to the session layer allows this code reuse. Also, there are far fewer session-layer networking protocols than application layer protocols; reengineering a few session-layer protocols is less expensive than trying to modify every application-layer networking protocol to perform the same functions.

Most session-layer security protocols already perform some kind of authentication. It is natural to add trust negotiation as an additional authentication facility to allow strangers to authenticate each other at the same level as the existing authentication code. Also, secure credential ownership verification (see Section 2.3) is most easily achieved using session-specific information that is frequently not available to applications.

It also makes more sense to modify existing session-level protocols than to create a new protocol for performing trust negotiation. Existing protocols address widely different networking needs. FTP is designed to perform file transfer between different computer architectures. SSH is designed for remotely accessing generic services, and

has provision for X11 forwarding. TLS is used to secure HTTP transactions. A new protocol created to accommodate trust negotiation, caching, and accumulation would have to support all of these features. It makes more sense simply to extend existing protocols.

## 1.5 Thesis Statement

This thesis discusses how existing session-layer security protocols can be extended to allow the authentication of strangers using automated trust negotiation, without reducing the protocols' security. Since trust negotiation can be costly in terms of computation and number of network rounds, authentication information is cached in the session-layer protocol to improve efficiency over the course of a session.

## 1.6 Summary of Upcoming Chapters

The remainder of the thesis is organized as follows. Chapter 2 contains a high-level discussion of some elements of cryptography and security that are relevant to the rest of the thesis. Chapter 3 describes research related to this thesis. Chapter 4 provides additional motivation for the thesis, and discusses trust caching and accumulation. Chapter 5 describes how trust negotiation can be incorporated into TLS without breaking it, and discusses caching and security issues. Chapter 6 does the same thing for SSH. Chapter 7 defines the messages for the two protocols in excruciating detail. Chapter 8 concludes the thesis and gives some ideas for future research.

(This Page Intentionally Left Blank)

# Chapter 2 — Background Material

Network security makes heavy use of cryptography. This chapter gives a brief overview of some types of cryptography and describes where they are used in trust negotiation and the session-layer protocols discussed in this thesis.

Security is often divided into three services: authentication, confidentiality, and integrity. Authentication means proving something about the sender of a message (the sender's name, for example, or that the sender is a police officer). Confidentiality means preventing anyone between a message's sender and recipient from reading the message. Integrity means preventing anyone from changing the sender's message without the knowledge of the recipient. A secure session protocol such as SSH or TLS supports all three services.

Symmetric-key cryptography is used to provide confidentiality, as discussed in Section 2.1. Public-key cryptography and digital credentials are used for authentication. Public-key cryptography is discussed in Section 2.2, and digital credentials are discussed in Sections 2.3 and 2.4. Mechanisms for proving integrity are not discussed here, because they are not referred to in the remainder of the paper.

Trust negotiation, TLS, and SSH make use of public-key cryptography and digital credentials. In order to limit access to sensitive information, trust negotiation also requires access-control policies, which are discussed in Section 2.5.

## 2.1 Symmetric-Key Cryptography

Symmetric-key cryptography is often used to provide confidentiality for messages sent across a network. In order to make the contents of a message incomprehensible to someone who can get access to the message in transit, the message is *encrypted*.

Encryption involves the use of a secret value, called a *key*, that is shared by the

sender and recipient of a message. (In symmetric-key cryptography, both the sender of the message and its recipient have the same key—hence the term "symmetric.") To encrypt the message, the key and message are given as input to a symmetric-key *cipher*, which performs a transformation on the input and outputs the *ciphertext*. The ciphertext looks like random data to anyone who doesn't have the key.

After the message is encrypted it is sent to the recipient. The recipient takes the ciphertext and the key and gives them as input to the cipher's inverse transform, which returns the message, bit-for-bit identical to the sender's original message.

Symmetric-key ciphers are designed to be very fast. That is, they are designed to have high throughput, encrypting large amounts of data very quickly. They are suitable for encrypting data even over fairly high-bandwidth connections. Since anyone who has the key can read any messages sent between the key's owners, the key must be carefully protected.

## 2.2 Public-Key Cryptography

Public-key cryptography is similar to symmetric-key cryptography in a few ways. They both use keys to perform transformations on data. Some methods of public-key cryptography can also be used for encryption. However, public-key cryptography is slow. Encrypting large amounts of data with public-key cryptography would be impractical.

Public-key cryptography differs from symmetric-key cryptography in that there are two different keys involved. One key is referred to as the private key; the other is the public key. The private key is used to decrypt data encrypted using the public key, and the public key is used to decrypt data encrypted using the private key.

For most usage scenarios, private keys and public keys are just that: private and public. The private key is carefully protected by its owner, but the public key is given out to anyone who wants it. To demonstrate why this is advantageous, consider the

following example, which contrasts key distribution using symmetric-keys and public-key cryptography.

**Symmetric Key Distribution** Suppose that there is a group of $n$ people. Each person in the group wants to have a secure connection with each of the others, so that nobody else can read messages sent between the two. Each person will need $n-1$ keys to preserve that confidentiality—one key shared with each other member of the group. Overall, this requires $\sum_{i=1}^{n-1} i$ keys, which is $(n^2 - n)/2$. For a large group, managing this many keys very quickly becomes a headache.

Contrast the number of symmetric keys required with the number of keys that public-key cryptography requires under the same scenario.

**Public Key Distribution** Each group member has a public and private key. Suppose that Bob wants to send a message to Alice. He finds Alice's public key and encrypts the message with it. Alice decrypts the message using her private key. Nobody else can read the message, because they don't have Alice's private key. For each person to have secure communication with everyone else, only $n$ key pairs are required (where a pair consists of a public key and its corresponding private key).

One problem with this example is that encrypting a message of any size using public-key cryptography is time-consuming and computationally expensive. This problem can be addressed by combining symmetric and public-key cryptography. If Alice wants to send a message to Bob, she can create a random symmetric key, encrypt the message with it, encrypt the symmetric key with Bob's public key, and send Bob the encrypted message and the encrypted symmetric key. Bob decrypts the symmetric key using his private key, and uses the symmetric key to decrypt the message. Since nobody else knows Bob's private key, Alice is guaranteed that nobody

between Alice and Bob can read the message.

The next problem is how Alice can prove to Bob that she sent the message. One approach is for Alice to encrypt the symmetric key with Bob's public key and then her private key. Then Bob would decrypt the symmetric key with Alice's public key and his private key, and use it to decrypt the message. Bob then knows that the message was from Alice because otherwise he would get incomprehensible bits instead of the message he was expecting.

More commonly, though, Alice computes a collision-resistant one-way hash on the message. Such hash functions return a fixed-size output. They are collision-resistant in the sense that it is difficult to find two blocks of data that have the same resultant hash. They are one-way functions in the sense that it is very difficult to determine the block of data that resulted in the hash from the hash alone. This allows the hash to be used as a data-integrity verifier. If one or more bits in the message is changed (purposefully or accidentally), the hash will fail to verify.

Alice *signs* the message message hash with her private key. Signing the hash has two purposes: it preserves the hash so that nobody can modify the message and expect the hash to compute, and it verifies that Alice sent the message.

So from start to finish, Alice computes a hash of the message, signs the hash with her private key, appends the signed hash to the message, creates a symmetric key, encrypts the message+hash with it, and encrypts the symmetric key with Bob's public key. She then sends Bob the ciphertext and the encrypted symmetric key. Bob decrypts the symmetric key using his private key, decrypts the ciphertext with the symmetric key, verifies the signature on the hash using Alice's public key, computes the hash on the decrypted message, and verifies that the computed hash and decrypted hash match. If they don't, he knows that either the message wasn't from Alice or that someone tampered with it. If the hashes match, he knows that the message is

from Alice and that it wasn't tampered with. PGP [3] uses this method to provide authentication, integrity protection, and confidentiality to e-mail.

Since session-layer security protocols deal with streams of data rather than asynchronous messages, they use public-key cryptography a little bit differently. However, session-layer security protocols (with the exception of Kerberos) similarly use public-key cryptography to agree on a symmetric key, symmetric-key cryptography for bulk encryption, and public-key cryptography for authentication.

The authentication method described in this section has another problem that must be addressed: How does Alice know that what someone claims is Bob's public key is really his? After all, Mallory could claim to be Bob and give Alice Mallory's own public key—then anytime Alice tries to send a message to Bob, Mallory can read the message, and Bob can't. One approach to solving this problem is to require exchange of public-keys to be face-to-face. This is largely impractical, and doesn't solve the problem if the people exchanging public keys are strangers. A better solution involves the use of digital credentials as discussed in the next section.

## 2.3 Digital Credentials

Digital credentials are analogous to the paper credentials used in the brick-and-mortar world. Many credentials contain attributes about the bearer. A driver license, for example, may contain the bearer's address, weight, height, driver license number, and so forth. A birth certificate contains the date and place of birth and parents' names. A house title may contain the location of the property and a description of the house. Just like paper credentials, digital credentials can contain attributes.

A credential is created by an *issuer* to state something about a particular *subject*. The subject is the owner of the credential. The issuer creates the credential by signing a hash of the body of the credential (the part that contains information about the issuer and subject) with his private key and affixing the signature to the end of the

credential. A recipient of the credential can verify that the credential is valid by obtaining the issuer's public key and verifying the signed hash against a hash of the message. If the hashes match, the credential was really signed by the issuer, since nobody else has the issuer's private key.

Digital credentials are frequently referred to in the literature as digital certificates. This usage has a long association with identity-based authentication (especially X.509 certificates). This thesis will refer to them as credentials except when describing an identity-based authentication system or defining terminology.

In the course of a negotiation, a negotiator, Charlie, sends the other negotiator a credential as proof that he (Charlie) has the attributes contained in the credential. Since it would be trivial for Charlie to present someone else's credential during a negotiation, there must be some provision made for proving ownership of the credential. Consequently, a digital credential either contains the subject's public key or a reference to the subject's public key. When the credential owner sends the credential to the other negotiator, the owner also presents some data that is unique to the session and known to both negotiators, signed with his private key. The credential recipient can then verify that the person presenting the credential is the one named in the credential by obtaining the owner's public key and verifying the data he received against his own knowledge of the session data. If the signature verifies, the credential presenter is the credential owner.

Trust negotiation makes the assumption that there is some well-defined method of encoding attributes so that disparate organizations agree on the interpretation of what the issuer means by putting a given attribute in the credential. The attributes encoded in the credential can then be mapped to a role on the local system.

Attributes contained in a credential may be sensitive. [10] discusses some methods of protecting and revealing sensitive attributes in digital credentials.

## 2.4 Credential Chaining

The previous section mentions that credentials are signed by a trusted third-party—the issuer. From the secret collaboration example (Section 1.2.2.1), if SecurityLeak receives a credential from a developer at Monopolysoft, SecurityLeak's server will scrutinize the credential to make sure that it is a valid credential. One of the first things that the server will check is whether or not it trusts the issuer.

To minimize the number of trusted issuers, credential issuing is frequently hierarchical. That is, one issuer issues a number of credentials to other issuers. For example, TrustCred, a fictitious credential issuing company, may generate credentials for both Monopolysoft and SecurityLeak (for a fee, of course). Monopolysoft can use the key associated with its credential to issue credentials to each of Monopolysoft's internal organizations. Each organization can issue credentials to its vice-presidents, managers, project leads, developers, and so forth. To verify that a developer's credential is valid, SecurityLeak's server verifies the signature on the credential, gets the credential's parent credential, verifies the signature on it, and so forth, on up to TrustCred's credential. Since SecurityLeak trusts TrustCred, the server knows that all of the credentials in the chain are valid. This group of credentials from TrustCred down to the developer is called a *credential chain*. In this example, TrustCred acts as a *root certificate authority* (root CA). TrustCred's credential is referred to as a *trusted root*. MonopolySoft's developer's credential is referred to as a *leaf credential*.

The foregoing discussion applies to normal identity-based authentication using public-key certificates. For trust negotiation the picture is a little more complicated. Each credential must be checked to verify whether the issuer is a valid issuer for the attributes contained in the credential.

## 2.5 Access-Control Policies

Access-control policies (or just policies) govern access to sensitive resources or services on a system. Historically, access control was encoded into whatever program or operating system controlled access to the resource. More recently, RBAC and other access-control methodologies have required the development of *policy neutral* systems: systems in which the access control policy or policies can be tailored to the resources that are being protected, and can be rapidly and easily updated as the access-control constraints change (for example, [14, 6]).

Trust negotiation puts some unique constraints on policy languages. To be useful for trust negotiation, a policy language must be able to make an access decision based on attributes of individuals trying to gain access. In order to reduce the network cost of trust negotiation, it must be able to export policies to the other negotiator in such a way that the other negotiator can evaluate the policy and verify that the policy belongs to the negotiator presenting it. The language must be monotonic—that is, the other negotiator can never lose privilege (access) by presenting more credentials. [17] presents additional facilities that a policy language must have to make trust negotiation feasible.

Policies may contain sensitive information, as discussed in the secret collaboration example of Section 1.2.2 above. [16] discusses an approach to deal with sensitive policies.

# Chapter 3 — Related Work

## 3.1 Session-layer Protocols

SSH2, the most recent version of SSH, is specified in [24], [22], [25], and [23]. [22] presents the general authentication protocol for SSH, and the public-key and username/password methods of identity-based access control. A description of SSH authentication is given in Section 6.1 of this document, and a comparison between SSH authentication and trust negotiation is given in Section 6.3.

TLS (specified in [4]) has optional client and server authentication based on public-key certificates. A description of TLS authentication is given in Section 5.1 of this document, and a comparison between TLS authentication and trust negotiation is given in Section 5.2. Most of the information in Chapter 5 was published previously [9], and contains results obtained from this thesis research. Chapter 5 contains some additional protocol details, modifications to the protocol presented, and information about caching and accumulation not present in the paper. Section 7.2 contains implementation details not present in the paper.

Kerberos [12] is a network authentication protocol that uses symmetric-key cryptography to authenticate users to services. At the completion of the protocol, the user and the service then have access to a shared secret key that can be used to encrypt session information. Authentication in Kerberos is identity-based. It is a possible venue for trust negotiation, in the same style as the modified SSH and TLS given in this thesis.

## 3.2 Identity-based Access Control

PGP ([3]) provides encryption and public-key authentication to e-mail (described in Section 2.2) and file systems. Authentication is purely identity-based. Because

PGP is intended for asynchronous messaging, it doesn't work with session-layer protocols.

## 3.3 Role-based Access Control

Role-based access control (RBAC) is described in Section 1.2. RBAC is an authentication system that maps users to roles, which are then mapped to permissions (see Fig. 1.1) to simplify administration and allow for finer-grained control over who has access to what. RBAC is in use in the corporate world. There are modules for many operating systems (including Windows 2000, Windows XP, Solaris, and Linux) that allow RBAC to be used to control access to file systems and system devices. [5] describes why previous access control models are insufficient and started NIST on the track to standardizing RBAC—an endeavor that is still in process.

DRIVE is gives an example of a distributed RBAC system designed for use in a health care environment [18]. DRIVE differentiates between static role assignment and dynamic allocation of roles using digitally-signed credentials. While the paper deals with a distributed environment, it assumes the existence of a single certifying authority, and uses credentials certifying role assignment rather than mapping attributes to roles from potentially different certifying authorities like trust negotiation does.

## 3.4 Trust Management

KeyNote is a trust management system used to provide flexible access-control [2]. It uses public-key cryptography in many of the same ways as trust negotiation. Trust management is a step forward from traditional identity-based authentication systems in the sense that it allows for constrained delegation of access rights. However, trust management assumes a closed system, and all authorizations are still identity-based. In contrast, trust negotiation is designed for authentication in open systems, and authorizations are based on attributes.

## 3.5 Trust Negotiation

This thesis builds on previous work in trust negotiation. The following papers describe this work.

Herzberg et al. [7] discusses the need for authenticating strangers over the Internet and presents the idea of authenticating strangers using attributes. The authors do not make the assumption of the existence of potentially sensitive credentials and policies, as trust negotiation does, so there is no negotiation involved. (This preserves perfect compatibility with existing session-layer authentication protocols.) The design includes a method of locating credentials necessary to create an authorization chain.

Winsborough et al. [19] considers how strangers can establish trust when the information used to establish that trust is sensitive. It is perhaps the seminal paper on trust negotiation, defining what it is and what the architecture for trust negotiation should look like. It also introduces the idea of negotiation strategies: a strategy defines in what order credentials and policies should be disclosed to the other negotiator. It assumes per-request authentication.

Yu et al. [26] gives an overview of issues in automated trust negotiation, including the architecture of automated trust negotiation, trust negotiation protocols, and negotiation strategies. It introduces the idea of strategy families for trust negotiation. It also describes how access control policies that contain sensitive information can be protected.

Policy information exchanged during trust negotiation may be sensitive, as discussed in Section 1.2.2, above. Seamons et al. [16] presents an approach to dealing with sensitive policies.

Seamons also discusses requirements that attribute-based authentication (trust negotiation) places on access-control policy languages [17].

(This Page Intentionally Left Blank)

# Chapter 4 — Session-Layer Trust Negotiation

## 4.1 Session-Layer Trust Negotiation

Previous research in trust negotiation has only considered single-access negotiation. That is, for each sensitive service or resource requested, trust negotiation occurs. Even if the same resource is requested multiple times, a trust negotiation occurs for each request. While this may work for HTTP and other connectionless protocols, it is clearly inefficient, and there are many session-layer network protocols where renegotiating for each requested service would be impractical at best, such as SSH, TLS, or FTP.

Session-layer trust negotiation (SLTN) improves on this by caching authentication information. Over the course of a negotiation, the authentication information accumulates. This process is described in the following two sections.

Research currently underway is examining the use of inter-session negotiation, wherein trust for a service is negotiated once and then can be resumed using a token that represents the negotiation that has already occurred. This differs from SLTN in two ways.

First, SLTN assumes a single session in which the client accesses a large number of services or resources that have similar policies. Inter-session trust negotiation considers trust resumption between interactions. Whereas inter-session trust negotiation requires the cryptographic verification of a token for every request, SLTN does not—it simply maintains previously negotiated trust for the duration of a session.

Second, inter-session trust tokens could potentially be delegated to provide transferable trust. This enables a single-sign-on style of authentication, where a single trust negotiation can be used to access services and resources on several servers. The

trust developed in SLTN, in contrast, is contained within the server that performs the authentications. Optimally, the two approaches can be combined so that there is a single cryptographic verification to resume all the trust established in the previous session.

## 4.2 Trust Caching

In SLTN, trust is *cached* for the duration of a session. Most session-layer protocols have a simple idea of trust caching that consists of the user logging in with a username and password and the application or operating system using that to determine access control for the remainder of the session. In role-based systems, the user is mapped to his set of authorized roles (which are stored locally on the server). When the user tries to access a service, his set of authorized roles is checked against the access-control policy for that service; if the policy is satisfied, the user is granted access. Since trust negotiation deals with situations where a user's attributes are more interesting than his identity, this simple mapping between a user and his authorized roles is insufficient. This idea will be explored further in the following sections.

## 4.3 Trust Accumulation

In traditional role-based systems, the mapping from user to role(s) can be done up-front when the user accesses a service, because the mapping is relatively inexpensive: Everything necessary to map a user to his authenticated roles is stored locally on the server or in a repository to which the server has access (for example, [18]). In trust negotiation systems, the up-front cost of mapping a user's attribute credentials to all of the possible roles he could satisfy could be quite expensive in terms of computation, network bandwidth, and number of rounds.

This is especially true if the user does not use all of the roles to which he is authorized in the course of the session, in which case much of the up-front computation is wasted. As a consequence, it is more efficient to address the role-mapping issue on

demand: a trust negotiation is performed for each service request the user makes for which he does not meet the access-control policy. Each additional role to which the user authenticates is added to the cache—hence the term trust *accumulation.*

## 4.4 Session-Layer Protocols

The following chapters describe how SLTN can be incorporated in two existing secure session-layer protocols: TLS and SSH. The implementations built from these analyses involve the use of a middleware implementation of trust negotiation called TrustBuilder, developed at Brigham Young University's Internet Security Research Lab. TrustBuilder performs the computation necessary to decide whether access to a secure resource or service should be granted, and decides which credentials and policies are released at each round of the negotiation. The session-layer protocols are modified to allow the transfer of this information. Other trust negotiation systems could implement everything at the session layer. (At present, to the best of my knowledge, no other implementations of trust negotiation exist.)

Please note that the difference in notational style used in the next two chapters is deliberate. The message and constant identifiers for each protocol are described using the same notation as their specifications.

(This Page Intentionally Left Blank)

# Chapter 5 — TLS Analysis

TLS is a connection-oriented protocol that provides a secure channel between a client and a server. TLS supports confidentiality, data integrity, and client/server authentication. The TLS handshake protocol provides a means for authentication and the negotiation of security parameters, such as the encryption algorithms, encryption keys, and so forth, that are used to transmit data securely. The TLS record protocol specifies how application data is actually transmitted between two communicating hosts to provide confidentiality and data integrity. Since trust negotiation deals with authentication, most of the following discussion will revolve around the handshake.

## 5.1 TLS Authentication

Client and server authentication in TLS are handled in the handshake protocol. This section describes the TLS handshake protocol for client/server authentication and identifies the limitations in the protocol for authenticating strangers on the Internet. As there are a large number of key exchange methods for TLS, this discussion is limited to the RSA key exchange method where the server requires mutual authentication. Using RSA key exchange with mutual authentication, the client and server exchange X.509 identity certificates. Only certain messages that are relevant to authentication will be explained here. For more information about the TLS handshake, refer to the specification [4] or Eric Rescorla's excellent book on TLS [15]. For clarity, the TLS handshake using RSA key exchange is depicted in Figure 5.1a.

The client initiates the handshake by sending a `ClientHello` message to the server. The server responds with a `ServerHello` message. These messages contain the necessary information to decide on an encryption cipher and integrity parameters for the TLS session.

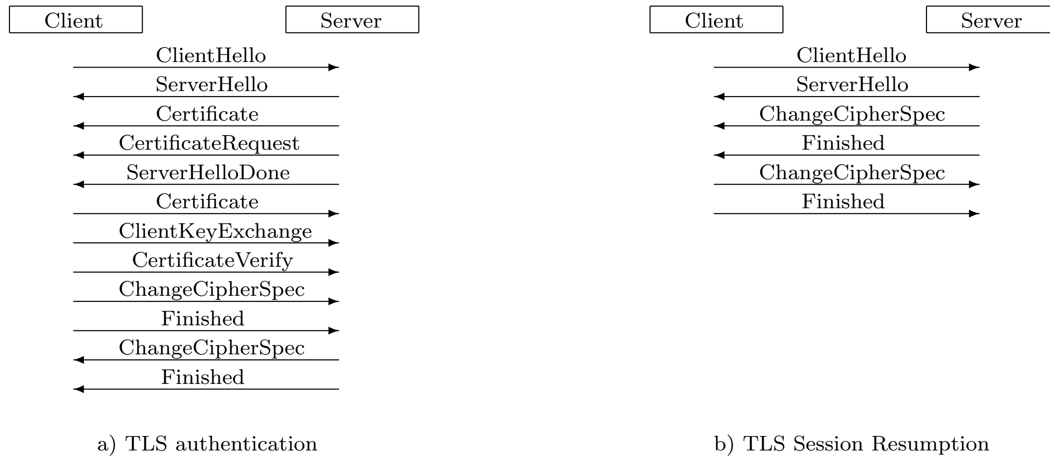


Figure 5.1: TLS messaging

The server continues the handshake by sending a `Certificate` message containing an X.509 certificate or certificate chain. The server then sends a `CertificateRequest` message, communicating the following three items of information to the client: first, that the server requires the client to authenticate itself by sending a certificate; second, a list of certificate types the server is willing to accept; and third, a list of X.500 distinguished names of certificate authorities (CA's) that the server trusts. The server specifies the certificate type as RSA. The list of trusted certificate authorities assists the client in selecting a certificate or certificate chain to submit that is signed by a root CA that the server trusts. Finally, the server sends a `ServerHelloDone` message indicating that it is now the client's turn to continue the handshake.

The next phase of the handshake protocol consists of messages sent from the client to the server. First, the client sends a `Certificate` message to the server containing an X.509 certificate or certificate chain. The next message is a `ClientKeyExchange` message, containing a client-generated pre-master secret to be used for key generation. The client encrypts the pre-master secret using the server's public key as contained in the certificate that the client received earlier in the handshake. The only way for the

server to successfully decrypt the message and obtain the pre-master secret is if the server possesses the private key associated with the certificate the server previously sent to the client. Thus, the `ClientKeyExchange` serves as an implicit challenge for the server to prove ownership of the private key. If the server successfully decrypts the pre-master secret, it can generate the correct keys to be used during the TLS session.

This phase of the handshake concludes when the client sends a `CertificateVerify` message to prove that the client possesses the private key associated with the certificate the client just sent to the server. The `CertificateVerify` message consists of a hash of all previous messages exchanged during the handshake, signed with the client's private key. The server decrypts the message using the public key contained in the client certificate and compares the result to a hash of all the previous messages exchanged during the handshake. If the decrypted `CertificateVerify` message matches the hash that the server has generated, the server knows that the client possesses the private key.

The handshake now enters the final phase. The client sends a `ChangeCipherSpec` message, indicating that the client will encrypt all further messages using the cryptographic keys that were just computed. Then the client sends a `Finished` message to the server containing a hash of all the preceding messages exchanged during the handshake, encrypted with the key shared between the client and the server. The server follows suit by sending its own `ChangeCipherSpec` and `Finished` messages to the client. Upon completion of the handshake, application data begins to flow through the secure channel. Note that no application data (such as a resource request) flows from the client to the server until after the encrypted session is established.

## 5.2 Limitations of TLS Authentication

TLS authentication is sufficient for certificate-based transactions for servers with a known user base. For example, Fifth Federal, a bank, could distribute X.509 certificates to each of its account holders, and configure its web server to require mutual authentication for secure banking transactions. An account holder could then install the certificate in his web browser and access the bank over a secure TLS connection as described in the foregoing discussion. However, TLS authentication is limited in the following ways:

1. Authentication is identity-based. It does not deal well with authenticating strangers.

2. Certificates are exchanged in plain text during the initial TLS handshake. While this does not allow an eavesdropper to misuse the certificate (for example, by masquerading as the client), exchanging certificates in the clear does introduce privacy risks when certificate contents are sensitive.

3. If the certificate chain received by either the client or the server does not completely satisfy their authentication requirements, there is no facility in the protocol for requesting additional certificates to meet all of the authentication requirements.

So, while TLS authentication is sufficient for the Fifth Federal example given in this section, it would not be sufficient for the scenarios given in the Introduction. Because TLS's authentication mechanism is identity based, it cannot perform attribute-based authentication. The client and the server are only allowed one credential presentation each, on a pass/fail basis. Incorporating trust negotiation into TLS solves these problems, as discussed in the next section.

## 5.3 Extending TLS to Support Trust Negotiation

Previous papers have suggested a way to incorporate trust negotiation into TLS [9]. This enhanced TLS is called "Trust Negotiation in TLS" (TNT). TNT leverages two parts of the existing TLS specification, the rehandshake and session resumption. Additionally, it relies on an Internet Draft under consideration to allow for generalized extensions to the TLS protocol. Adding trust caching and accumulation promises to make the system more efficient.

The analysis presented here differs slightly from what we proposed previously. TLS only provides for one certificate type in the key exchange: X.509. Trust negotiation may involve other credential types. This analysis adds a few messages to the protocol given in the TNT paper to allow for the exchange of additional credential types during trust negotiation. It also describes how a client can initiate trust negotiation in the case that the client application wants to push some sensitive data to the server. Changes from the TNT paper are noted as they are described.

### 5.3.1 TLS Rehandshake

Once a TLS connection is established using the handshake protocol described in Section 5.1, it is possible to conduct a TLS rehandshake. The rehandshake is simply the TLS handshake performed over an existing TLS connection. Either the client or server can initiate a rehandshake. A client initiates a rehandshake by sending a new `ClientHello` message to the server after a previous handshake has finished. A server initiates a rehandshake by sending a `ServerHelloRequest` message to a client. Upon receipt of a `ServerHelloRequest` message, the client responds with a `ClientHello` message, and the handshake continues as usual.

Since trust negotiation may involve sensitive certificates, negotiation must be confidential. During TLS client/server authentication in an initial TLS handshake, certificates are exchanged in plain text. To ensure that credential and policy contents

are confidential in TNT, trust negotiation only occurs during a rehandshake initiated during an encrypted TLS session.

### 5.3.2 TLS Session Resumption

According to [15], the performance bottleneck in the TLS handshake is the public-key cryptographic operations. In particular, the encryption and decryption required to exchange a pre-master secret is expensive. One of the reasons the client must verify the server's certificate is so that it can use the server's public key to encrypt the pre-master secret in the key exchange. TLS provides session resumption as a way to avoid the overhead of a full TLS handshake. With session resumption, an abbreviated handshake occurs as follows (depicted in Fig. 5.1b).

The client sends a `ClientHello` message to a server and includes the SessionID from a previous session with the server. If the server is willing to resume the session, the server replies by returning the same SessionID in the `ServerHello` message. In order to resume a session, the client and server reuse the master secret from the prior session to compute new keying material, thus avoiding the expensive public-key operations of a normal handshake. After the `ServerHello` message they simply exchange `ChangeCipherSpec` and `Finished` messages, with the server proceeding first.

TNT leverages TLS session resumption in order to avoid the overhead of needlessly generating a new master secret. Once the client and server successfully negotiate trust, an abbreviated handshake takes place, similar to session resumption. Instead of completing the full handshake, the client and server compute new keying material by reusing the master secret from the previous TLS session and the random nonces exchanged in the `ClientHello` and `ServerHello` messages.

### 5.3.3 TLS Message Extensions

The Transport Layer Security working group is developing a backward-compatible mechanism for extending TLS to make it useful in environments where it was not originally envisioned: For example, TLS was not designed to be used by wireless devices that have physical constraints such as low bandwidth connections or limited memory. The working group has proposed [1] to make TLS more flexible. The draft was just accepted as an RFC in June 2003, so it is evident that there is some enthusiasm for incorporating it into the TLS standard.

The TLS extensions allow a client to propose a list of extensions to be used during the TLS handshake. It does this by adding a list of extensions to the `ClientHello`. This does not break TLS, because the TLS specification makes explicit provision for the client doing so. (However, the TLS specification does not delineate how such extensions are to be formatted or recommend any extensions; the draft does.) An extension consists of a two-byte identifier, the value of which determines the extension that the client is requesting, and an opaque string of data consisting of a two-byte length field and the actual data.

When the server receives an extended `ClientHello`, it has the option of returning a list of extensions in the `ServerHello`. The extended `ServerHello` may not contain any extensions that were not present in the `ClientHello`. The next section describes how trust negotiation uses these extended `-Hello` messages.

### 5.3.4 Facilitating Trust Negotiation in TLS

The following describes how TNT overcomes the limitations enumerated in Section 5.2. Figure 5.2 is provided to clarify this discussion.

Trust negotiation may be initiated by either negotiator. The server initiates trust negotiation when the client requests a resource for which the client has not yet met the server's access-control policy. The client requests trust negotiation when the client

application is ready to send some sensitive data to the server for which the server has not yet satisfied the client's access-control policy. The negotiator who holds the resource for which trust is being negotiated will be referred to as the *initiator* for the remainder of this discussion. The other negotiator will be called the *respondent*.

If the server is the initiator of the trust negotiation, the negotiation proceeds as follows. After the initial TLS handshake, when a client presents a resource request to the server, the server checks its access-control policy, determines that the resource requested is sensitive, and initiates a trust negotiation rehandshake, modeled after the TLS rehandshake described in Section 5.3.1. This is done by sending a `HelloNegotiationRequest` message to the client. (Server-initiated trust negotiation is depicted in Figure 5.2.) If the client's TLS does not support trust negotiation, upon the receipt of the `HelloNegotiationRequest` message the client's TLS will abort the connection with an `Alert` message containing the constant `unexpected_message`. This is not a problem, since the client cannot meet the server's access-control policy.

If the server requests trust negotiation as described in the previous paragraph or the client has secure data to send to the server, the client sends the server a `ClientHello` message containing two extensions: one that tells the server that the client is prepared to negotiate trust (the trust negotiation extension), the body of which is empty; and one that contains a list of trust negotiation strategy families the client is willing to use.

If none of the client's proposed strategy families is acceptable, the server aborts the trust negotiation by sending an `Alert` message containing the constant `trust_negotiation_failed` (given in Section 7.2.3). Otherwise, the server replies with a `ServerHello` message containing the trust negotiation extension and one strategy family selected from the client's list of proposed strategy families. (The client should ensure that the server's selection was in the list that it proposed.)

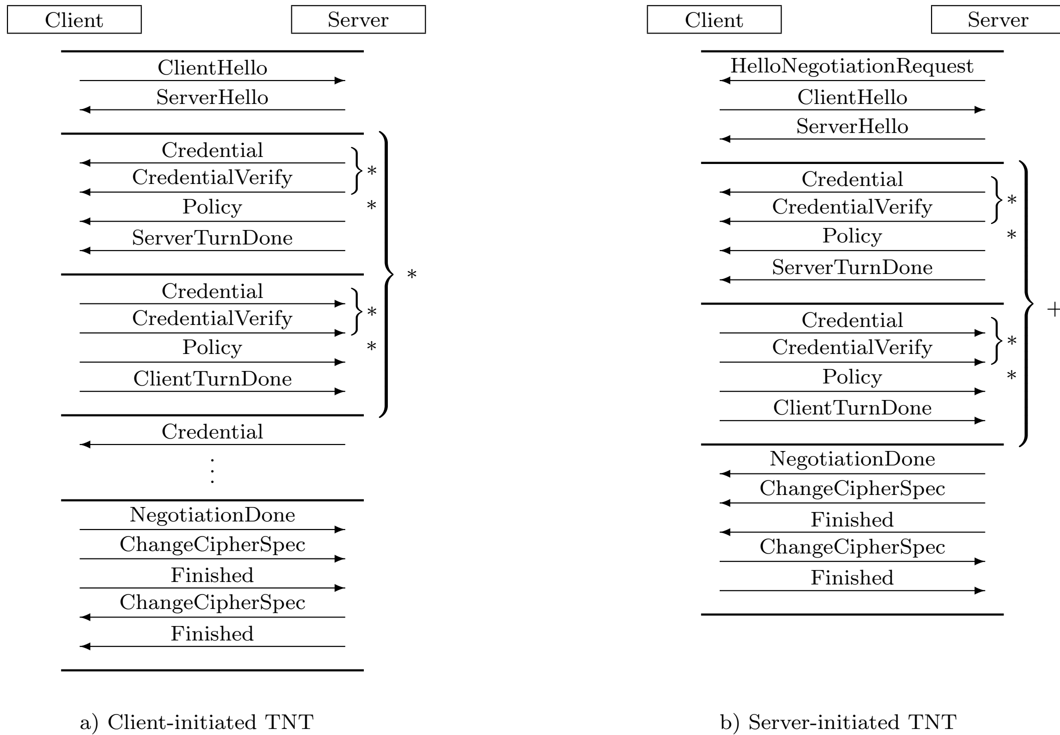


Figure 5.2: TNT messaging. * indicates a message or set of messages that may occur zero or more times in succession. + indicates a set of messages that may occur one or more times in succession.

The client and server then proceed to negotiate trust using six messages added for that purpose: `Credential`, `CredentialVerify`, `Policy`, `ServerTurnDone`, `ClientTurnDone`, and `NegotiationDone`. (This is slightly different from [9]. In [9] we proposed using the TLS messages `Certificate` and `CertificateVerify`. These messages cannot be used for trust negotiation, because the TLS messages assume that the contained certificates are X.509 certificates. Trust negotiation allows for a more flexible range of credential types.)

The server begins the first round of negotiation, as described in the following paragraphs. Thereafter, the negotiators take turns sending sets of credentials and policies until the initiator either grants or denies access to the respondent, or the respon-

dent terminates the negotiation. The server and client send `ServerTurnDone` and `ClientTurnDone` messages, respectively, to indicate when they are finished sending policy and credential information for a particular round of negotiation. Note that the `Server-` and `Client-` in the `-TurnDone` messages refer to the negotiators' TLS roles rather than their roles in the trust negotiation. The `-TurnDone` messages contain no data; they are used exclusively for flow control. The set of credential and policy information a negotiator sends during a round will be referred to as his "turn."

During a negotiator's turn, any number of `Credential` and `Policy` messages may be sent. All `Credential` and `CredentialVerify` messages must precede the first `Policy` message for the turn.

Each `Credential` message contains a two-byte field that specifies the credential type, a three-byte length field that specifies the length of the credential body, and the body of the credential, itself.

When a negotiator sends a credential that he owns and the public key in the credential has not yet been verified, a `CredentialVerify` message must immediately follow the `Credential` message. The `CredentialVerify` contains a digital signature, encoded the same as for the `CertificateVerify` message already defined in TLS. The data that is signed in a `CredentialVerify` is slightly different from the data that is signed in a `CertificateVerify`. A `CertificateVerify` message contains the signed hash of all of the previous handshake messages sent during the current handshake. A `CredentialVerify` contains the signed hash of all the previous handshake messages (for the current handshake) up until the most recent `-TurnDone` message received from the other negotiator. Since there is no "most recent `-TurnDone` message" the first time the server sends its turn, the `CredentialVerify` contains the signed hash of the `ClientHello` and `ServerHello` messages from this handshake.

Each `Policy` message contains an access-control policy that the other negotiator

must satisfy. The message consists of a three-byte length field, followed by the actual policy data.

A negotiator may choose not to send any `Credential` or `Policy` messages during his turn. If one negotiator sends no `Credential` messages and no `Policy` message during his turn, the other negotiator's strategy will decide whether or not to terminate the negotiation.

When the respondent satisfies the initiator's access-control policy, the initiator sends a `NegotiationDone` message, followed by `ChangeCipherSpec` and `Finished` messages. The respondent responds with `ChangeCipherSpec` and `Finished` messages. The initiator can then deliver the resource to the respondent.

### 5.3.5 Negotiation Failure in TLS

TLS provides two levels of `Alert` messages: fatal and non-fatal. Fatal `Alert`s invariably terminate the TLS session. TLS clients and servers may or may not terminate the TLS session upon receipt of a non-fatal `Alert`.

Trust negotiation failures fall into both categories. If one of the negotiators sends an invalid credential or policy, for example, the other negotiator can assume that the negotiator who sent the bad data is not negotiating in good faith and terminate the session by sending a fatal `Alert` message. Appropriate errors for invalid credentials include `bad_certificate`, `unsupported_certificate`, `certificate_revoked`, `certificate_expired`, `certificate_unknown`, and `unknown_ca`. The TLS specification describes the circumstances under which each of these errors will be sent. Errors for invalid policies include `bad_policy`, which is sent when a policy has invalid formatting or a signed policy fails to verify; `unsupported_policy`, which is sent when a negotiator receives a policy type it does not recognize; and `policy_unknown`, which indicates that there is some unspecified error in the policy. The policy `Alert` messages are not defined in the TLS specification; standard TLS does not involve policy

exchange. The additional constants that TNT adds to support these `Alert`s are given in Section 7.2.3.

When trust negotiation fails because the respondent does not meet the initiator's access-control policy, the initiator should send a non-fatal `Alert` to the respondent, containing the constant value `trust_negotiation_failed`. The session is still valid, and further resource requests can be made, but the secure resource for which trust was being negotiated must not be released.

## 5.4 Backward Compatibility

TNT is backward compatible with TLS. There are eight cases to consider for compatibility, as summarized in the following table.

| | Client | Server | Negotiation Required | Result |
|---|---|---|---|---|
| 1. | TLS | TLS | No | Works |
| 2. | TLS | TLS | Yes | Will never occur |
| 3. | TLS | TNT | No | Works |
| 4. | TLS | TNT | Yes | Fails cleanly |
| 5. | TNT | TLS | No | Works |
| 6. | TNT | TLS | Yes | Fails cleanly |
| 7. | TNT | TNT | No | Works |
| 8. | TNT | TNT | Yes | Works |

Table 5.1: TLS/TNT Compatibility

The first, second, seventh, and eighth cases do not concern us here, since they don't involve compatibility between TLS and TNT. It is simple to demonstrate that the other four cases work or fail appropriately, as we would expect.

The initial handshake is identical for TLS and TNT, so the fifth case will succeed whenever a normal TLS handshake would succeed. The design discussed in the previous section does not tamper with the normal TLS rehandshake facility, or with normal TLS session resumption, so these will behave normally, too.

When a TLS client attaches to a TNT server and requests a resource that doesn't require any authentication other than the normal TLS handshake, the server will return the resource without requesting trust negotiation, so the third case succeeds as expected. The third and the fifth cases are sufficient to demonstrate backward compatibility. For completeness, the following paragraphs discuss upward compatibility.

When a TLS client attaches to a TNT server and requests a resource that does require trust negotiation, the TNT server sends the client a `HelloNegotiationRequest`. The client will not recognize the message type, and will send a fatal TLS `Alert` message. This will terminate the session. Because the session terminates in a handshake failure, it is not resumable. This is unfortunate, but unavoidable.

The sixth case occurs when a TNT client attaches to a TLS server. They request and respond merrily along until the client wants to push some sensitive content to the server that the client is not willing to release without an appropriate level of trust in the server, as described in an earlier thesis [8]. The client then initiates a trust negotiation by sending a `ClientHello` with the appropriate header extensions, as discussed in Section 5.3.4. If the server does not know how to deal with header extensions, the server ignores the extensions and treats the rehandshake as a normal rehandshake, sending a normal `ServerHello`. The client may either terminate the TLS session when this occurs or continue the session and not send the data, because the server obviously cannot meet the client's access-control policy.

## 5.5 Caching and Accumulation

TLS has some special issues with regard to caching, because one of its primary uses is to secure HTTP requests. Since HTTP is a stateless protocol, connections last as long as a single request. (Since HTTP 1.1, several requests/responses may be transferred over the same TCP connection. However, either the client or the server may break the connection after any request/response pair, regardless of the

logical session.) In order to accommodate the possibility of several connections per session, TLS makes use of a session identifier as described in Section 5.3.2. If further authentication is necessary, the client and server can perform a complete rehandshake (without resuming the session). However, the application would have to cache the additional authentication information, because TLS does not.

If the server is configured to allow session resumption, when a TLS connection terminates the server stores the session identifier and the master secret for the session, along with the cryptographic parameters negotiated for the session. If the client does not return to the server using the SessionID within some unspecified period of time (the TLS specification recommends one day), the session state expires from the server's cache. In TNT, all of the authentication information is stored along with the session information that is relevant to resumption.

The client stores the same information, so that when the user returns to the server the client knows to present the SessionID to the server. TLS provides no way for the client to know when the session state will expire from the server's cache. It isn't really necessary for the client to know—when the server demands that the client reauthenticate, the client reauthenticates.

When the TLS session is removed from the server's cache, the server should also remove the negotiated authentication state (that is, authentication information about the other negotiator). It certainly does not make sense to retain the authentication state for any longer than the TLS session, since the only way the client can reference the authentication state is through the SessionID. (If it is valuable to retain authentication state for a longer period of time, the server and client should use the inter-session trust negotiation techniques alluded to in Section 4.1.) Removing the authentication state before the TLS session state runs the risk of inefficiency: If the client resumes the session after the authentication state expires but before the

TLS session state expires, the client and server will have to renegotiate to reestablish all of the authentication state that just expired. Since TLS session resumption was designed to avoid unnecessary, expensive cryptographic operations, expiring the authentication state before the TLS session state expires seems senseless.

No additional provision needs to be made to allow for trust accumulation in TNT: the application already has an interface to request renegotiation. After the client has met the access-control policy for the initial resource request, the session continues until the client tries to access another resource (or service) for which it has not already met the access-control policy. At that point, the server application calls down to TNT, requesting renegotiation for the sensitive resource that the client tried to access. Negotiation occurs as described in the previous section, and any additional authentication information is added to the session cache.

## 5.6 Security

Security is a difficult thing to prove, both in design and in implementation. This section will therefore demonstrate the security of TNT by arguing that the modifications made to TLS do not reduce its security.

Because trust negotiation is only performed in a rehandshake, after the initial handshake has already completed, TNT cannot reduce the security of the initial handshake. The initial handshake establishes keying material to provide encryption, host-based authentication, and data integrity services to every message that follows. This means that nobody between the client and the server can modify or insert messages in the stream without being detected.

Every TLS message is tied to the session that was established in the initial handshake, so an attacker cannot replay messages later in order to gain access to secure resources on either the client or the server side. Furthermore, an attacker cannot

perform a man-in-the-middle attack.[1]

All trust negotiation credential ownership verifications are tied to the TLS session, so the credential ownership verifications are also not subject to replay or reflection attacks.

The most significant potential security hole that trust negotiation introduces to TLS is the set of access-control policies used to guard access to secure resources. Experience has shown that even simple scenarios must be carefully tuned so that those who should get access to a secure resource can, and anyone else cannot. Writing secure access-control policies is an appropriate area for further research.

Also, granting access to unknown users on a system is always a security risk. Trust negotiation is precisely that—a negotiation to establish trust. Obviously some precautions must be taken to limit the range of actions that an authenticated client can take, so that their permissions are appropriate to the level of trust they have developed. Policies play a large part in this. One approach to dealing with this problem is to have some kind of auditing system that can sweep the system and determine whether permissions are set appropriately. Another valuable way to audit the system is to log any suspicious activity, including the public keys, roles, and credentials of the authenticated user who is performing the action. If an exploration of the log divulges evidence of wrong-doing, the public key or keys associated with the credentials of the wrong-doer can be blacklisted.

[1] This is only partly true. TLS provides a way to have anonymous (unauthenticated) servers. This is subject to a man-in-the-middle attack, so it is not usually recommended. TNT actually mitigates this problem, because the server must still satisfy the client's access-control policies to gain access to anything the client considers sensitive. This does, however, ruin the anonymity.

## Chapter 6 — SSH Analysis

SSH provides applications encryption, authentication, and data integrity services. SSH is intended to run over a reliable transport protocol, such as TCP. There are two versions of SSH, imaginatively called SSH1 and SSH2. Use of SSH1 is deprecated because of some security problems, so this discussion will only describe and analyze SSH2. From here on, "SSH" will refer to SSH2.

SSH consists of three logical layers, called protocols, that build on each other. Each protocol provides an application different services (see Table 6.1). The layers, from lowest (closest to TCP) to highest, are the *transport* protocol, the *authentication* protocol, and the *connection* protocol.

The transport protocol establishes an encrypted, integrity-protected session between the client and the server. It also provides for host-based server authentication. The authentication protocol is then used to prove that the client is acting on behalf of a user who is allowed access to the server—this provides user (client) authentication. The connection protocol allows several applications to share a single authenticated session, multiplexed over the same logical socket.

| Protocol Layer | Description |
|---|---|
| Transport | Provides encryption, integrity-protection, and host-based server authentication to authentication and connection protocols and applications. |
| Authentication | Provides user authentication to connection protocol and applications. |
| Connection | Multiplexes several channels so that they can all share the same transport and authentication layers. Each channel is a separate application (or service). |

Table 6.1: SSH Protocol Layers

The authentication protocol is of primary interest to this discussion. The following

sections describe how SSH provides for a number of authentication methods, and why the specification as it is currently defined is not sufficient to support trust negotiation and accumulation.

## 6.1 SSH Authentication

SSH is designed to make it easy to add new authentication methods. There are several methods already defined by which the client can prove that he is allowed access, including username/password, host-based, and public-key authentication.

After a secure connection is set up between the client and the server using the transport protocol, the client requests a service using the `SSH_MSG_SERVICE_REQUEST` message[1] [25]. `SSH_MSG_SERVICE_REQUEST` contains a service name, which is simply a string.[2] Certain services are directly specified; all others are assumed to abide by the format "service-name@domain.name".

The SSH specification explicitly defines two services that are available immediately after the transport protocol completes: "ssh-userauth", and "ssh-connection". The "service-name@domain.name" style of referencing a service provides for additional implementation-specific services or extensions. By requesting "ssh-userauth" or "ssh-connection", the client has the option of requesting client authentication or requesting an unauthenticated connection, respectively. If the client requests the connection service directly over the transport protocol (i.e. an unauthenticated connection), he may only request services that allow unauthenticated usage.

The "ssh-userauth" service provides an authenticated tunnel over which the "ssh-connection" service (or some other locally-defined service) may be requested. The "ssh-connection" service provides a logical connection over which the client may request a number of execution channels, which are multiplexed over the single connec-

[1] All SSH messages that are described in this chapter are defined in Section 7.3.2.

[2] The data types used to describe messages in this chapter are defined in Section 7.3.1.

tion. If the client requests the "ssh-connection" service immediately after finishing the transport protocol, the client will be denied any service requests that require authentication. If the client requests "ssh-connection" immediately after completing the authentication protocol (i.e. "ssh-userauth"), execution of client requests is decided based on the authentication information the client provided during the authentication protocol.

## 6.2 SSH User Authentication

The client initiates every authentication interaction. Authentication is performed with respect to a particular service. It is permissible for the server to require or allow several forms of authentication—for example, the server could require both password authentication and public-key authentication for access to a particular service.

Figure 6.1 illustrates user authentication at a high level to clarify the discussion that follows. The content of each message is given above (or around) the arrow representing the message. To keep the illustration concise, the `SSH_MSG_` prefix is left off the constants that identify the messages.

After the client and server complete the transport protocol, the client requests the "ssh-userauth" service, and the server responds with an `SSH_MSG_SERVICE_ACCEPT` message. The client then makes a request for a single form of authentication by sending an `SSH_MSG_USERAUTH_REQUEST` message:

The `SSH_MSG_USERAUTH_REQUEST` message contains three pieces of information: the user name, the service to start when authentication completes, and the authentication method the client wants to use. The rest of the message depends on the authentication method. For example, in the password method of authentication, the message also contains the user's password. Some authentication methods can result in further messages being exchanged that are specific to the authentication method.

If the client sends the server an `SSH_MSG_USERAUTH_REQUEST` message with the au-

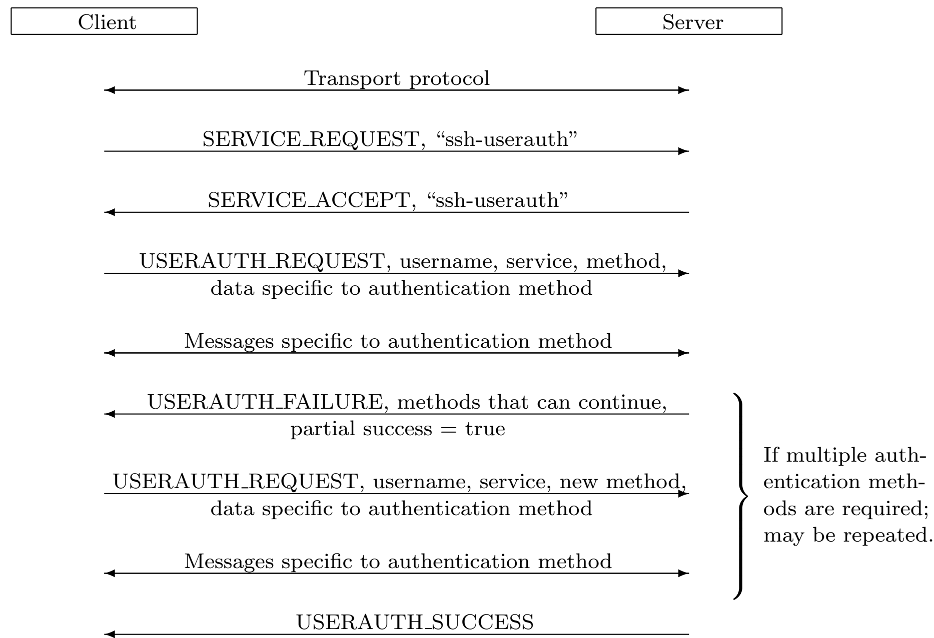


Figure 6.1: SSH user authentication

thentication method set to "none", the server sends the client a list of supported authentication methods. While it seems like the client could use this list to determine how it can authenticate to gain access to the resource, it is not required that any of the authentication methods that the server returns be relevant to accessing the requested resource.

At the completion of the requested authentication method, the server has two options. If the username does not exist or the client fails to authenticate, the server can either disconnect or send an `SSH_MSG_USERAUTH_FAILURE` message.

This message contains a flag to indicate partial success, which is set to true if the client successfully authenticated using the requested method, but the method was not sufficient to gain access to the service. For example, if a particular service requires public-key authentication, but the client successfully completed password authentication, an `SSH_MSG_USERAUTH_FAILURE` message would be returned with the partial

success flag set. On failure to authenticate, it is allowable for the client to continue attempting to authenticate using other methods (unless the server disconnects, of course—then the client must start all over).

If the client successfully completes sufficient authentication to gain access to the requested service, the server returns an SSH_MSG_USERAUTH_SUCCESS message. The server will ignore any further SSH_MSG_USERAUTH_REQUEST messages after sending SSH_MSG_USERAUTH_SUCCESS.

## 6.3 Limitations of SSH Authentication

1. A username is always required as part of the authentication mechanism.

2. Authentication is always client-requested. Trust accumulation requires that the server be able to request further authentication, to determine if the client can gain access to resources for which he has not already satisfied the access-control policy.

3. Authentication can only be performed once for a given authentication layer. That is, once authentication is complete, authentication may not be requested again: "SSH_MSG_USERAUTH_SUCCESS MUST be sent only once. When SSH_MSG_USERAUTH_SUCCESS has been sent, any further authentication requests received after that SHOULD be silently ignored" [22]. This is especially problematic for trust accumulation, which is not possible without some way to request further authentication information.

4. Authentication is only performed at the beginning of a session, and access to any resources requested during the session is determined solely on that authentication. If any resources' access-control policies are not met by the authentication information gathered during this initial authentication, the client is disallowed access—even if the client has the credentials necessary to access those resources.

5. Server authentication is assumed to be performed in SSH transport protocol, and is purely host-based. No attribute-based server authentication is performed. Because the authentication layer runs on top of the transport layer, there is no way for a client to build trust iteratively using SSH's authentication mechanisms.

6. The client has no certain way to discover the server's authentication policy with respect to a particular resource.

## 6.4 Extending SSH to Support Trust Negotiation

There are two ways to overcome the limitations expressed previously. One involves modifying the SSH authentication protocol to allow for trust negotiation's requirements. The other is to define a new service that can be requested after the transport protocol completes. The difference between the two options is small but significant.

Modifying the existing protocol has the disadvantage that it requires modifying the existing protocol. Existing code that implements the protocol would no longer be valid. However, since SSH2 is still in draft with the IETF, it is still possible to make changes to the protocol without too much difficulty. Further, since it *is* in draft, those organizations that have coded SSH2 have done so with the understanding that it is a moving target and their existing code may have to be revised.

Adding trust negotiation as a new service has the advantage that it does not require modifying the existing protocol. The new trust negotiation service could be started immediately after the transport protocol completes, after the authentication protocol completes, or at any time during the execution of a service. (Starting trust negotiation after completion of an identity-based authentication method doesn't make much sense, though. Since trust negotiation is designed for open systems and all the other authentication methods are designed for closed security domains, the two

methods likely wouldn't be used simultaneously. While an organization could have the policy that only users known to the system who have certain attributes can have access to a service, this policy can be implemented using standard RBAC. Since the computation of rights in standard RBAC is less computationally intensive than trust negotiation and consumes less network bandwidth, such an organization would more likely use standard RBAC for this type of access control.)

On the other hand, adding trust negotiation as a new service means that it is much more difficult to use it in conjunction with the authentication methods that are already designed for SSH. Specifically, it is difficult for the client to determine whether it needs to authenticate using identity-based authentication or trust negotiation to gain access to a particular resource. In general, however, the user will know whether or not he has an account on the server from which he is requesting a service. Another option is for the client simply to try both.

In keeping with the previously-stated objective of preserving backward compatibility, and in light of the difference between trust negotiation and SSH's existing authentication protocols, it seems better to add trust negotiation as a new service rather than modifying the existing user authentication layer. This results in two authentication protocols that coexist at the same layer of SSH, as depicted in figure 6.2. To differentiate between the two, the normal SSH user authentication layer will be referred to as the authentication layer or authentication protocol, and the trust negotiation authentication layer will be referred to as the trust negotiation layer or trust negotiation protocol.

The trust negotiation protocol may occur at any time after completion of the transport protocol. Either the client or the server may initiate the trust negotiation. It is desirable for the client to initiate a trust negotiation under two circumstances: first, the client can initiate trust negotiation as the first service request after the

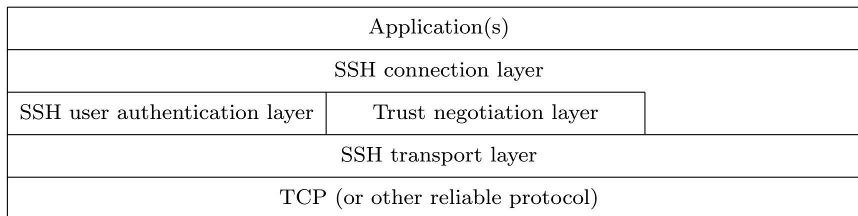


Figure 6.2: SSH with trust negotiation

transport protocol completes; second, if the client wants to push sensitive content to the server, the client can request trust negotiation before sending the data. See [8] for more details. The server initiates trust negotiation when the client requests a resource for which he has not yet met the access-control policy.

### 6.4.1 Requesting Trust Negotiation as the First Service

To request authentication using trust negotiation after completion of the transport protocol, the client requests the "trust-negotiation@isrl.cs.byu.edu" service by sending an `SSH_MSG_SERVICE_REQUEST` message.

If the server supports trust negotiation and is willing to negotiate, it returns `SSH_MSG_SERVICE_ACCEPT`. Otherwise it returns `SSH_MSG_DISCONNECT`. (This is the same behavior as described in [24], to preserve backward compatibility.)

If the client receives `SSH_MSG_SERVICE_ACCEPT`, it continues the negotiation as in Section 6.4.2 below.

Following completion of the trust negotiation protocol, if the client does not have sufficient permissions to start the service it requested (see the explanation of the service field in `SSH_NEGOTIATION_REQUEST` in Section 6.4.2), the server may either simply drop the connection—TCP FIN, for example—or send the client an `SSH_MSG_DISCONNECT` message and then drop the connection.

### 6.4.2 Facilitating Trust Negotiation in SSH

SSH uses a single byte to differentiate between message types. To minimize the number of message type values that adding another protocol consumes, a single message identifier is used to identify trust negotiation messages: `SSH_MSG_TRUST_NEGOTIATION`. Each message then contains another byte identifier that specifies what type of negotiation message it is. Figure 6.3 shows the process of trust negotiation in SSH.

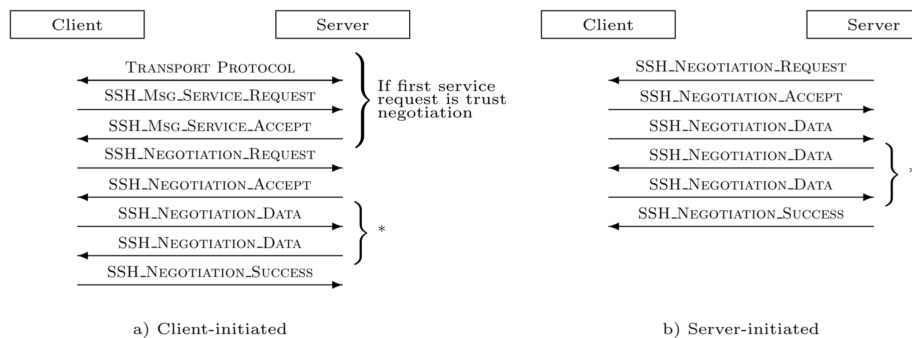


Figure 6.3: Trust negotiation in SSH. * denotes a set of messages that may repeat zero or more times.

Trust negotiation begins with two messages that allow for strategy family selection (see Section 1.2.3). The initiator of the trust negotiation proposes one or more strategy families, and the other negotiator (the respondent) selects one.

The initiator makes the proposal by sending an `SSH_NEGOTIATION_REQUEST` message. This message contains two items of information: a *service* string, and a name-list containing a list of trust negotiation strategy families that the client supports. In the case that the client is requesting trust negotiation immediately after completion of the transport protocol, the service is the service that the server should start after completion of the trust negotiation protocol. If the client only desires a single authenticated service, the service name is the name of the service to be started. If the client may request multiple services over the authenticated channel, the "ssh-connection"

protocol may be started after trust negotiation completes.

In the case that the negotiation request is sent when a service has already started, the service string identifies the resource for which access is being negotiated. This is informational; negotiation strategies can use this identifier to decide when to send which credentials and policies and when to terminate the negotiation. (Canonicalization of the identifier is not considered in this thesis.) This service string may be empty—zero length. This is desirable when the respondent should not know what is being negotiated until it is established that he has certain attributes, as described in [8], for example.

The name-list specifies the trust negotiation strategy families that the initiator is willing and able to use, most preferred first.

The respondent should ignore any proposed strategy families that it does not recognize. If none of the initiator's proposed strategy families are acceptable to the respondent, the respondent should send `SSH_NEGOTIATION_FAILURE` with the reason code set to `NO_COMMON_STRATEGY_FAMILY`.

`SSH_NEGOTIATION_FAILURE` contains two fields: a reason code and a printable explanation string. The reason code gives the standardized reason that the negotiation failed. The explanation field contains more information about the failure. It may be displayed to help the failing negotiator know how to correct the problem, so the recipient should take proper precautions to ensure that the string really is printable and of a reasonable length, as described in [24].

The `SSH_NEGOTIATION_FAILURE` message differs from `SSH_MSG_USERAUTH_FAILURE` in two ways. First, if none of the authentication methods succeed when using the authentication protocol, the server terminates the session. With the exception of the first client service request after the transport protocol completes (as described in Section 6.4.1), `SSH_NEGOTIATION_FAILURE` has no such implication. If a session is already established

and a service started, it is expected to continue after negotiation failure, because the negotiators have already established enough trust in each other to create the session and run a service. This is not to say that access should be granted to the resource for which trust negotiation failed. Access to resources protected under an access-control policy *must not* be granted when trust negotiation fails. However, when negotiation fails, the application has the option of whether to continue the session or abort it.

Second, `SSH_MSG_USERAUTH_FAILURE` is only sent by the server. An `SSH_NEGOTIATION_FAILURE` message may be sent by either negotiator.

If one or more of the initiator's proposed strategies are acceptable, the respondent should select the first strategy family that it is also capable and willing to use. The respondent then sends the initiator an `SSH_NEGOTIATION_ACCEPT` message that specifies the family that the respondent selected. `SSH_NEGOTIATION_ACCEPT` may also contain credential and policy information for the first round of negotiation. For some strategies it is either not desirable or not possible for the respondent to send negotiation information first.

Every negotiation message sent by either negotiator after the strategy selection messages is either an `SSH_NEGOTIATION_DATA` message or a negotiation terminating message (`SSH_NEGOTIATION_FAILURE` or `SSH_NEGOTIATION_SUCCESS`). The `SSH_NEGOTIATION_DATA` message contains all of the credential and policy information relevant to this round of the negotiation.

A negotiation ends in success or failure. How to deal with failure is described above. If the respondent meets the access-control policy for the service under negotiation, the initiator sends `SSH_NEGOTIATION_SUCCESS`.

Only the initiator is permitted to send `SSH_NEGOTIATION_SUCCESS`. If the initiator receives the success message, it must terminate the negotiation and return failure to the application that requested negotiation. It may also drop the connection, since

the respondent is either non-compliant or malicious.

In the case that the negotiation occurs immediately after completion of the transport protocol, successful negotiation results in the server starting the service that the client requested in the SSH_NEGOTIATION_REQUEST. Otherwise, successful completion of the negotiation returns success to the application, which may then return the resource or implement some higher-level authorization policy (based on server load or quality of service constraints, for example).

### 6.5 Backward Compatibility

The modifications to SSH that are described in Section 6.4 are backward compatible with SSH as it is presently specified. This argument was made at the beginning of that section. Trust negotiation is defined as a separate service from SSH's other user authentication methods. Connection-layer services (including "ssh-connection") use the trust negotiation layer as though it were the authentication layer. This allows for perfect backward compatibility.

If the user has an account on the server, trust-negotiation-enabled SSH (TNE-SSH) clients can request identity-based authentication on normal SSH servers by requesting the "ssh-userauth" service after completing the transport protocol. Authentication continues as described in the SSH specification.

If the user does not have an account on the server, TNE-SSH clients can request access using the "trust-negotiation@isrl.cs.byu.edu" service. If the server does not support trust negotiation, it will not recognize the client's service request and simply disconnect, sending an SSH_MSG_DISCONNECT message with the reason code set to SSH_DISCONNECT_SERVICE_NOT_AVAILABLE. From the server's point of view, compatibility is preserved: the server simply does not provide the service that the client requested.

Finally, a normal SSH client can request any identity-based services that a TNE-SSH server provides. As long as the user has a login on the server, he will get access.

## 6.6 Caching and Accumulation

SSH does not have the clever session resumption feature that TLS has. Consequently, SSH session management is much simpler. An SSH session begins when the client and server complete the transport protocol. It ends when either the client or server drops the TCP connection, or when one of them sends an `SSH_MSG_DISCONNECT` message. Authentication state is stored for the duration of the session.

Whereas in TLS key exchange and authentication are part of the same process (the handshake), in SSH rekeying is performed at the transport layer and authentication is performed by using either the authentication protocol or the trust negotiation protocol.

The authentication protocol has no need for any kind of reauthentication to obtain more authentication information, so authentication is only performed at the beginning of the session. Further, authentication is always client-requested.

In order to provide for trust accumulation, the trust negotiation protocol makes explicit provision for reauthentication in the form of trust negotiation. Either the client or the server may request additional authentication information at any time (although some cautions are in order; see Section 7.3.3 for more details).

## 6.7 Security

Like the security analysis in the TLS chapter (Section 5.6), this section argues that TNE-SSH is no less secure than the original SSH specification.

The transport protocol provides encryption, data integrity, anti-replay, and host-based server authentication to SSH clients. Since trust negotiation is only allowed after the transport protocol completes, the trust negotiation protocol does not interfere with the transport protocol in any way. The only concerns that trust negotiation add to the system are the last two described in the TLS chapter: poorly-written access-control policies and allowing strangers access to the system.

(This Page Intentionally Left Blank)

# Chapter 7 — Implementation Notes

In order to demonstrate that it is feasible to build trust negotiation into session-layer security protocols, the protocols described in Chapters 5 and 6 were implemented. Both implementations use TrustBuilder, a middleware trust agent that performs the necessary computations to negotiate trust.

## 7.1 TrustBuilder

TrustBuilder is under development at the Internet Security Research Lab at Brigham Young University. TrustBuilder acts as a trust agent for one negotiator—that is, it manages keys, credentials, and policies for him. It implements trust negotiation strategies to determine which credentials and policies should be released at any point during a negotiation.

TrustBuilder is written in Java. It can be accessed directly from a Java program, or indirectly through a SOAP connection. Currently, TrustBuilder uses IBM's TrustEstablishment system to map attributes contained in X.509 credentials to local system roles. Another system is under development that uses RTML credentials [13].

## 7.2 TLS

TNT is implemented in Claymore Systems' open-source Java version of TLS, PureTLS. Section 7.2.1 contains an overview of how negotiation proceeds using TNT, from the application's perspective. Section 7.2.2 contains an explanation of TLS data types. Section 7.2.3 contains definitions for the messages TNT adds to TLS.

### 7.2.1 TNT Negotiation Overview

The server runs an HTTP server with sensitive resources served over a TNT SSLServerSocket. The client's browser attaches to the server using a TNT SSLSocket (at which point the server's SSLServerSocket returns an SSLSocket). The client's

SSLSocket performs a TLS handshake with the server, establishing a secure connection. The browser requests the resource over the socket. The HTTP server looks up the resource, determines that it is sensitive, sets the access-control policy in the server's SSLSocket's policy information and tells it to renegotiate using the trust negotiation extensions. The server's SSLSocket requests trust negotiation. The client's socket accepts, and the two implementations negotiate trust (possibly involving multiple rounds of negotiation). The server's SSLSocket accepts the client's credentials as sufficient to allow release of the resource, notifies the client that negotiation succeeded, and returns success to the HTTP server. The HTTP server delivers the resource to the client.

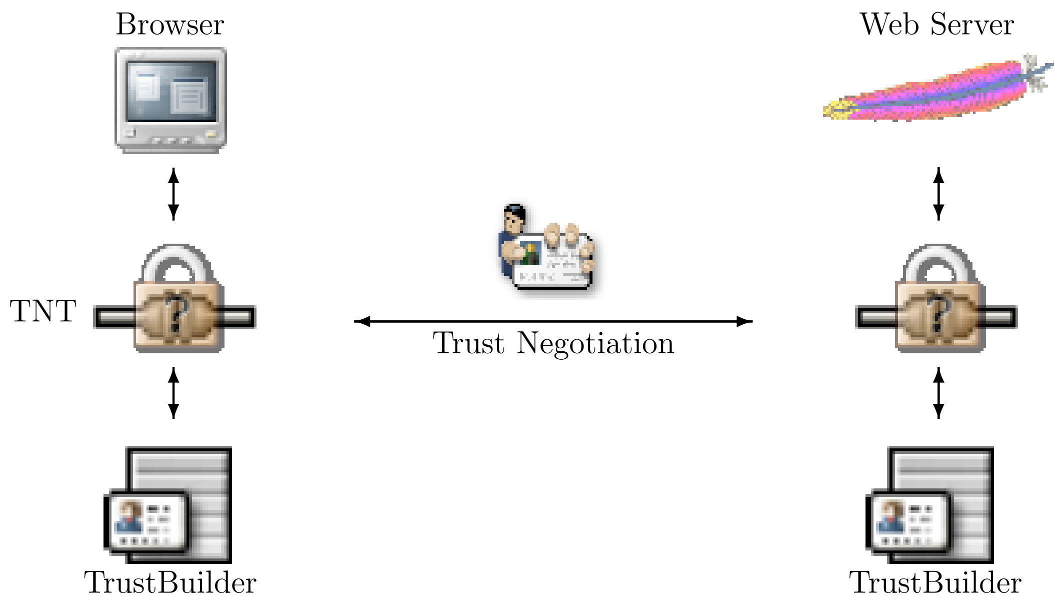


Figure 7.1: TNT architecture

### 7.2.2 TLS Data Types

The TLS specification, [4], describes messages using the following description language. Only those data types that are necessary to describe the additional messages created for TNT are described here.

**Opaque** "Opaque," when used to describe a value, means that TLS does not interpret the data.

**Vectors** A TLS vector is a one-dimensional array of a single data type. Sizes are always given in bytes, not in number of elements. For example, suppose that a data type `IceCreamCone` consists of a fixed-size array of thirty-one `Flavor` elements, and each `Flavor` is four bytes:

```
opaque  Flavor[4];
Flavor  IceCreamCone[124];
```

Variable length vectors use angle brackets, which contain a range of valid lengths, like so:

```
opaque  Pounds<200..150000>;
```

Variable length vectors are encoded by prepending the data with a length field that consists of the least number of bytes that can represent the maximum value of the range. For example, a `Pounds` type that is 350 bytes long would be encoded as a three-byte length field (since three bytes is sufficient to encode 150000) containing the value 350, followed by the 350 bytes of the `Pounds` object.

Vectors must be an integral multiple of the base type. A 130-byte `IceCreamCone` would be illegal, because an `IceCreamCone` is constructed of `Flavor`s, and each `Flavor` is four bytes.

**Numeric Values** The fundamental data type in TLS is the octet: an eight-bit value. All other data types are derived from octets. The TLS specification represents an octet with `uint8`. Four other integral data types are built from the `uint8`:

```
uint8  uint16[2];
uint8  uint24[3];
uint8  uint32[4];
uint8  uint64[8];
```

All multi-byte integral values are in network order (big-endian).

**Enumerations** Enumerations are almost C-style `enum`s, except the enumeration type name follows the definition instead of preceding it. For example, an enumeration of main courses for dinner might be as follows:

```
enum  { MacAndCheese(0), Steak(1), Fish(2), Lasagna(3) (255) }  MainCourse;
```

Each element of the enumeration has two parts: a name and a value. The name is the name by which a particular value in this enumeration can be referred. The name is followed by a value enclosed in parentheses. The value is what is actually sent on the wire when the enumeration is sent. (Enumerations that are not sent across the wire may or may not have a value defined.) The last element of the `MainCourse` enumeration is optional; if present, it gives the maximum value for the enumeration. This allows the size of the enumeration to be set larger than the value of largest defined element, to allow for future growth. The type identifier for the enumeration (`MainCourse`) follows the rest of the definition.

**Structures** A `struct` (called a "constructed type" in the specification) is a C-style `struct`: it contains other data types in sequence.

```
struct {
   Flavor IceCreamCone[124];
   opaque Pie;
} Dessert;
```

The various elements of the `struct` can be referred to by name: `Dessert.Pie`.

**Variants** A variant is a `struct` with one or more fields that depend on system state. For example, a `Dinner` variant might be as follows.

```
enum { Salad(0), Soup(1) } SideDish;
enum { false(0), true(1) } Diet;
```

```
struct { } Empty;

struct {
   SideDish   appetizer;
   MainCourse dinner;
   select (Diet) {
      case false: Dessert;
      case true:  Empty;
   } dessert;
} Dinner;
```

Every element of the `enum` on which the `select` selects must be defined in the selection. The selection name (`dessert`) might not be present.

**Cryptographic Signature** Among other cryptographic identifiers, the TLS specification defines `digitally-signed`, which simply indicates the signature over the `struct` data that follows. A `digitally-signed` field is encoded as:

```
opaque  digitally-signed<0..2^16-1>;
```

### 7.2.3 TNT Message Definitions

This section contains definitions for the messages that TNT adds to TLS, as described in Section 5.3.4 in the syntax described in the previous section. The `Signature` and `SignatureAlgorithm` data types are defined in [4]. They are provided here as a convenience, since the `CredentialVerify` uses them.

```
struct { } HelloNegotiationRequest;
struct { } ServerTurnDone;
struct { } ClientTurnDone;
struct { } NegotiationDone;

opaque Policy<0..2^24-1>;

enum { X.509(0), RTML(1), (65535) } CredentialType;

opaque CredentialBody<0..2^24-1>;

struct {
   CredentialType type;
   CredentialBody body;
```

```
} Credential;

enum { anonymous, rsa, dsa } SignatureAlgorithm;

select (SignatureAlgorithm) {
   case anonymous: struct { };
   case rsa:
      digitally-signed struct {
         opaque md5_hash[16];
         opaque sha_hash[20];
      };
   case dsa:
      digitally-signed struct {
         opaque sha_hash[20];
      };
} Signature;

opaque SignatureType<0..65535>

struct {
   SignatureType type;
   Signature signature;
} CredentialVerify;
```

The `SignatureAlgorithm` for the `CredentialVerify` data type is determined by the type of the signature key contained in the `Credential` message that immediately precedes it. Since there are several methods of signing with each key type, the `SignatureType` field of `CredentialVerify` is included to disambiguate exactly which signature method was used.

TNT also adds some `Alert` messages to TLS. For clarity, they are presented here alongside the other TLS `Alert` enumerated values and the `Alert` message itself.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
   close_notify(0),
   unexpected_message(10),
   bad_record_mac(20),
   decryption_failed(21),
   record_overflow(22),
   decompression_failure(30),
   handshake_failure(40),
   bad_certificate(42),
   unsupported_certificate(43),
```

```
   certificate_revoked(44),
   certificate_expired(45),
   certificate_unknown(46),
   illegal_parameter(47),
   unknown_ca(48),
   access_denied(49),
   decode_error(50),
   decrypt_error(51),
   export_restriction(60),
   protocol_version(70),
   insufficient_security(71),
   internal_error(80),
   user_canceled(90),
   no_renegotiation(100),
   bad_policy(110)
   unsupported_policy(111)
   policy_unknown(112),
   trust_negotiation_failed(120),
   (255)
} AlertDescription;

struct {
   AlertLevel level;
   AlertDescription description;
} Alert;
```

The `ClientHello` and `ServerHello` extensions for requesting trust negotiation are defined here. They are presented alongside the existing extensions `enum` for clarity, and the extended `ClientHello` and `ServerHello` are provided for your information. The other fields of the `Client-` and `ServerHello` are described in [4].

```
struct {
   ProtocolVersion client_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suites<2..2^16-1>;
   CompressionMethod compression_methods<1..2^8-1>;
   Extension client_hello_extension_list<0..2^16-1>;
} ClientHello;

struct {
   ProtocolVersion server_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suite;
   CompressionMethod compression_method;
   Extension server_hello_extension_list<0..2^16-1>;
```

```
} ServerHello;

struct {
   ExtensionType extension_type;
   opaque extension_data<0..2^16-1>;
} Extension;

enum {
   server_name(0), max_fragment_length(1),
   client_certificate_url(2), trusted_ca_keys(3),
   truncated_hmac(4), status_request(5),
   trust_negotiation(6), strategy_families(7),
   supported_signature_methods(8),
   (65535)
} ExtensionType;

strategy_families Extension {
   ExtensionType strategy_families;
   Family families<0..2^16-1>;
};

supported_signature_methods Extension {
   ExtensionType supported_signature_methods;
   opaque signature_method<0..2^16-1>;
};

enum { Eager(0), DTS(1), TTG(2), (65535) } Family;
```

## 7.3 SSH

TNE-SSH is implemented in J2SSH, a Java implementation of SSH2 under development by the open-source community.

Section 7.3.1 contains an explanation of SSH data types. Section 7.3.2 contains definitions for the messages TNE-SSH adds to SSH. Section 7.3.3 contains a few final considerations in implementing TNE-SSH.

### 7.3.1 SSH Data Types

Data types for SSH messages are described in [24]. Only types that are used in the following discussion are given in Table 7.1.

Integral constant values (represented by integral values) are given in all capitals, like this: `DEMONSTRATION_CONSTANT`. Constant strings are represented in quotes, like this:

byte — An 8-bit value.

boolean — A byte; zero indicates false, and anything else is true, although the specification mandates that compliant implementations may only store the value 1 for true.

uint32 — A 32-bit integral value, stored in network byte order (most significant byte first).

string — Arbitrary data (not necessarily printable). The first four bytes of a string are a uint32 specifying the length of the string, which is followed by the string data. Strings are not null-terminated (except by coincidence). They may be zero length, in which case no data follows the length field.

name-list — A comma-delimited list of names, represented as a string. Names are encoded in UTF-8 [21], and may not contain a comma (for obvious reasons). For example, a list with two names, "name1" and "name2", would be represented as follows (in hexadecimal notation):

```
Length field  ‘‘ n  a  m  e  1  ,  n  a  m  e  2’’
00 00 00 0b      6e 61 60 65 31 2c 6e 61 60 65 32
```

Table 7.1: SSH Data Types

"demonstration string". A list of constants for SSH—both from the original specification and added for trust negotiation—can be found in Appendix A.

### 7.3.2 TNE-SSH Messages

This section gives definitions of both the existing SSH messages that are relevant to the analysis of Chapter 6 and the additional messages that are necessary to allow for trust negotiation. Messages are described using the datatypes given in the previous section. A short explanation follows each message, describing the message's purpose and fields.

#### 7.3.2.1 Existing SSH Messages

```
byte    SSH_MSG_SERVICE_REQUEST
string  service name
```

The `SSH_MSG_SERVICE_REQUEST` message is used to request a service after the transport protocol, authentication protocol, or trust negotiation protocol completes. Two service names are defined in the SSH specification: "ssh-userauth" and "ssh-connection." TNE-SSH adds a third, "trust-negotiation@isrl.cs.byu.edu." It is anticipated that if trust negotiation is added to the SSH specification, the "@isrl.cs.byu.edu" will be dropped. The "xxx@yyy.yyy.yyy" naming convention is used to represent local extensions.

```
byte    SSH_MSG_USERAUTH_REQUEST
string  user name
string  service name
string  authentication method name
  ⋮     method-specific data
```

An SSH client sends an `SSH_MSG_USERAUTH_REQUEST` message to the server when requesting identity-based authentication. The `user name` field is required, and contains the name of the user trying to authenticate to the system. The `service name` is the name of the service that the server will start after authentication, if authentication succeeds. The `authentication method name` specifies how the client wants to authenticate (for example, using password authentication or public-key authentication). Finally, the message contains fields that depend on the authentication method. For example, if the authentication method is "password", the message also contains the user's password.

```
byte     SSH_MSG_USERAUTH_FAILURE
string   authentication methods that can continue
boolean  partial success
```

The server sends the client `SSH_MSG_USERAUTH_FAILURE` either when the client's attempt to authenticate failed, or the client has not presented the right kind of authentication information to gain access to the service he requested. The `string` contains a comma-separated list of authentication methods that the client can still use to try to authenticate. (The difference between this type of string and a name-list is unclear—

but the specification lists the field as a string, so there you go.) The `partial success` flag indicates whether the client's last attempt to authenticate succeeded or failed.

The server may require more than one form of authentication to allow access to certain services, and does so by returning `SSH_MSG_USERAUTH_FAILURE` with the `partial success` flag set after the client successfully completes a portion of the authentication policy.

```
byte  SSH_MSG_USERAUTH_SUCCESS
```

The server sends this message to the client when the server accepts the client's authentication information as sufficient to obtain access to the service the client requested.

#### 7.3.2.2 Messages Added by TNE-SSH

```
byte    SSH_MSG_SERVICE_REQUEST
string  ``trust-negotiation@isrl.cs.byu.edu''
```

This is the service request that a client can send to the server after completing the transport protocol to request trust negotiation. The quotes are not part of the actual string.

```
byte       SSH_MSG_TRUST_NEGOTIATION
byte       SSH_NEGOTIATION_REQUEST
string     service (resource)
name-list  negotiation strategy families
name-list  supported signature methods
```

`SSH_NEGOTIATION_REQUEST` is sent by either the client or the server to initiate a trust negotiation. The `service` field contains the service that the server will start if trust negotiation succeeds, or an identifier that identifies the resource for which access is being negotiated. The `service` field may be empty. The `negotiation strategy families` field contains a comma-separated list of strategy families that the initiator supports, most preferred first. The `supported signature methods` field contains a comma-separated list

of signature methods that the negotiator supports.

| | | |
|---|---|---|
| | byte | SSH_MSG_TRUST_NEGOTIATION |
| | byte | SSH_NEGOTIATION_REQUEST_ACCEPTED |
| | string | negotiation strategy family selected |
| | name-list | supported signature methods |
| | boolean | negotiation data follows |
| | —— if boolean is true —— | |
| | uint32 | # of verified credentials to follow ($v$) |
| $v \times$ | string | verified credential type |
| | string | verified credential |
| | boolean | verification follows |
| | string | credential verification (if boolean is true) |
| | uint32 | # of unverified creds to follow ($u$) |
| $u \times$ | string | unverified credential type |
| | string | unverified credential |
| | uint32 | # of policies to follow ($p$) |
| $p \times$ | string | policy |

`SSH_NEGOTIATION_REQUEST_ACCEPTED` is sent by the respondent to notify the initiator that the respondent is willing to negotiate trust. The respondent also has the option of providing the first negotiation data in this message.

The `negotiation strategy family` string contains the name of the strategy family that the respondent selected from the initiator's proposed list. The initiator should verify that the selected strategy family was in the proposal. The next field of the message is a flag that specifies whether the message also contains negotiation data. If true, it indicates that the respondent's first round of negotiation data follows the flag. This could reduce the number of required messages by one, to improve efficiency. However, for some strategies it is either not desirable or not possible for the respondent to send negotiation information first, in which case the flag is set to false.

Negotiation data (if any) begins with an unsigned 32-bit integer that specifies the number of verified credentials that follow. A verified credential is one that the sender owns (that is, a credential for which he has the private key and can prove ownership).

The data for each verified credential contains the credential type, the credential

itself, a boolean flag that indicates whether verification data follows the credential, and verification data if the flag is true. The credential type specifies both the credential format and the encoding style (if applicable). For example, a DER-encoded X.509 credential is represented by the string “X.509-DER”. A PEM-encoded X.509 credential is represented by the string “X.509-PEM”. An RTML credential (see [13]) is the encoding style for RT credentials, so the credential type is simply “RTML”.

Verification data must follow the credential if the sender has not yet sent verification data for the public key contained in the credential during the current session. If the sender has already sent verification data for the public key in the credential, verification data may be sent, but it is not required. If verification data is present, the recipient must verify the data, even if the public key has already been verified.

If the verification data is absent, the sender must set the boolean verification data flag to false, and the recipient must ensure that the public key contained in the credential has already been verified. If the public key contained in the credential has not been verified, the recipient must send `SSH_NEGOTIATION_FAILURE` (see below) with the reason code set to `VERIFICATION_DATA_MISSING`.

If the credential has verification data associated with it (if the verification flag is true), then a string containing verification data follows the flag. The verification data consists of the session identifier established in the transport protocol, signed by the private key that corresponds to the public key contained in the credential.

Following the verified credential information is an unsigned 32-bit integer containing the number of unverified credentials to follow. Unverified credentials are credentials that are not owned by the sender, but are required to prove attributes of the sender. For example, a child’s digital birth certificate may contain information identifying his parents. When the child’s mother goes to prove that she is the child’s mother, she presents the child’s credential. Since the credential is not hers, however,

she cannot prove ownership of it.

The unverified credential information consists of two pieces of information: the credential type and the credential itself. The credential type is the same as for verified credentials.

After the unverified credential information is an unsigned 32-bit integer containing the number of policies that follow. Each policy is simply a string.

| | | |
|---|---|---|
| | byte | SSH_MSG_TRUST_NEGOTIATION |
| | byte | SSH_NEGOTIATION_DATA |
| | —— if boolean is true —— | |
| | uint32 | # of verified credentials to follow ($v$) |
| $v \times$ | string | verified credential type |
| | string | verified credential |
| | boolean | verification follows |
| | string | credential verification (if boolean is true) |
| | uint32 | # of unverified creds to follow ($u$) |
| $u \times$ | string | unverified credential type |
| | string | unverified credential |
| | uint32 | # of policies to follow ($p$) |
| $p \times$ | string | policy |

All of the fields in the SSH_NEGOTIATION_DATA message are as described for the SSH_NEGOTIATION_REQUEST_ACCEPTED message above.

```
byte  SSH_MSG_TRUST_NEGOTIATION
byte  SSH_NEGOTIATION_SUCCESS
```

SSH_NEGOTIATION_SUCCESS indicates successful completion of the trust negotiation. That is, the initiator accepts that the client is authorized to receive the resource. Only the initiator of the trust negotiation may send this message.

```
byte    SSH_MSG_TRUST_NEGOTIATION
byte    SSH_NEGOTIATION_FAILURE
uint32  reason
string  explanation
```

SSH_NEGOTIATION_FAILURE indicates that trust negotiation has failed. It may be sent by either negotiator. The reason code gives the standardized reason that negotiation

failed. Reason code are listed in Table A.3.

### 7.3.3 TNE-SSH Implementation Considerations

The interface between the negotiation layer and the connection layer allows any channel to request renegotiation. Any number of channels could request renegotiation simultaneously. Access to the negotiation sublayer is synchronized, so that multiple potentially overlapping negotiations do not collide, which could result in duplication of effort or corruption of data. Corollary to this, all channels reference the same authenticated roles for a particular user, just like how channels share identity-based authentication information in normal SSH.

(This Page Intentionally Left Blank)

# Chapter 8 — Conclusion and Future Work

## 8.1 Conclusion

As described in the Introduction (Section 1.5), the main contributions of this thesis are to demonstrate that trust negotiation can be incorporated into session-layer security protocols without breaking the existing protocols, and to discuss the implications of caching the trust accumulated during a session. Adding trust negotiation as an additional authentication facility makes the protocols more widely useful: Trust negotiation allows strangers to authenticate each other, which is difficult with the identity-based authentication protocols that currently exist in the protocols. For example, the open-source firewall example (Section 1.3.2) is possible using SSH extended to support trust negotiation. Using traditional SSH, it would be difficult.

Furthermore, adding trust negotiation to session-layer protocols makes trust negotiation more efficient in the case that resources with similar or identical access-control policies are requested during a single session. In the open-source firewall example, a script to remove a large number of files sequentially from the repository would run with just one trust negotiation, rather than negotiating for every file removal.

It has been shown by protocol analysis that trust negotiation can be incorporated into two existing session-layer security protocols: TLS and SSH. The modifications do not interfere with the security properties of either protocol. The resultant trust-negotiation-enabled protocols are also fully backward compatible with the existing protocols. This allows for gradual adoption of trust negotiation between cooperating parties on the Internet. Because the protocols are backward compatible, there is no need for a universal simultaneous upgrade like IPv6 requires, removing another barrier to the adoption of trust negotiation.

To demonstrate that the proposed protocol modifications are practicable, the modifications have been implemented in PureTLS and J2SSH. These serve as suitable reference implementations, as would be required for standardization by the IETF.

The thesis also contains a discussion of caching issues with respect to each protocol, and proposes when authentication information should expire from the cache. In general, it is argued that authentication information should expire from the session cache when the session-layer protocol's session does.

## 8.2 Future Work

There are other ways to improve the efficiency of trust negotiation in session-layer protocols. One such modification would be to cache when a user fails to authenticate to a particular role. Suppose that Mary from the credit-card example (Section 1.3.3) attaches to Fifth Federal's server and requests a credit card for her neighbor's child, Bobby. Mary and Fifth Federal negotiate trust, and it is determined that Mary does not have the credentials necessary to authenticate to the roles that allow her to get a credit card for Bobby. Suppose that Mary then tries to open a savings account for Bobby, and that opening a savings account on behalf of a minor has the same policy as obtaining a credit card. With caching as described in this thesis, Mary and Fifth Federal negotiate trust again. This is unnecessary, because they have already negotiated trust for those roles and determined that Mary does not meet them.

It is desirable to limit the amount of data encrypted by a single key for two reasons [11]: First, cryptographic keys are sensitive to the amount of data they encrypt—the more data a key is used to encrypt, the more subject the key is to compromise. Second, periodic rekeying limits the cost of a key being compromised if perfect forward secrecy is provided by the key establishment protocol: Only the portion of the session that was encrypted by the compromised key can be recovered by an attacker. Most established cryptographic session protocols already have some mechanism by which

rekeying can be performed, and recommend a period at which rekeying should be performed (usually based on bytes encrypted or time elapsed). It may be desirable to rekey more frequently when the client authenticates to a more sensitive role. For example, a distributed medical system might rekey more often for doctors viewing their patients' medical records than for medical researchers gathering summary statistics. The session-level protocols may or may not provide an interface by which the rekeying frequency can be set on a per-role basis. It would be valuable to define an interface to allow this kind of control.

(This Page Intentionally Left Blank)

## Appendix A — SSH Constants

A number of constants are defined for SSH. This appendix contains a list of some pre-existing SSH constants, as well as the constants that are defined for the trust-negotiation-enabled SSH in the thesis. It is not a complete list of SSH constants. Only those constants that are referenced in this thesis are given.

The byte value that identifies the type of the an SSH message is partitioned as follows [24]:

| Values | Description |
|---|---|
| 1–19 | Universal transport protocol messages |
| 20–29 | Messages for negotiating what key-exchange method to use |
| 30–49 | Messages that depend on the key exchange method (numbers can be reused by each key exchange method) |
| 50–59 | Universal user authentication messages |
| 60–79 | Messages that depend on the authentication method (numbers can be reused by each authentication method) |
| 80–89 | Universal connection protocol messages |
| 90–127 | Messages related to channels |
| 128–191 | Reserved |
| 192–255 | Local extensions |

Table A.1: Partitioning of the Byte Value Message Identifier for SSH

| Explanation Identifier | Value | Where Defined |
|---|---|---|
| SSH_MSG_SERVICE_REQUEST | 5 | [25] |
| SSH_MSG_SERVICE_ACCEPT | 6 | [25] |
| SSH_MSG_USERAUTH_REQUEST | 50 | [22] |
| SSH_MSG_USERAUTH_FAILURE | 51 | [22] |
| SSH_MSG_USERAUTH_SUCCESS | 52 | [22] |
| SSH_MSG_TRUST_NEGOTIATION | 200 | Here |
| SSH_NEGOTIATION_REQUEST | 1 | Here |
| SSH_NEGOTIATION_REQUEST_ACCEPTED | 2 | Here |
| SSH_NEGOTIATION_DATA | 3 | Here |
| SSH_NEGOTIATION_FAILURE | 4 | Here |
| SSH_NEGOTIATION_SUCCESS | 5 | Here |

Table A.2: Constant Values for SSH Messages

| Failure Code | Value |
|---|---|
| NO_COMMON_STRATEGY_FAMILY | 1 |
| VERIFICATION_DATA_MISSING | 2 |
| INVALID_CREDENTIAL | 10 |
| UNSUPPORTED_CREDENTIAL_TYPE | 11 |
| CREDENTIAL_REVOKED | 12 |
| CREDENTIAL_EXPIRED | 13 |
| CREDENTIAL_ERROR | 14 |
| UNKNOWN_CA | 15 |
| INVALID_POLICY | 20 |
| UNSUPPORTED_POLICY_TYPE | 21 |
| POLICY_ERROR | 22 |

Table A.3: Constant Values for Negotiation Failure in SSH

## Bibliography


[1] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (TLS) extensions. IETF RFC 3546, June 2003.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system version 2. IETF RFC 2704, September 1999.

[3] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. IETF RFC 2440, November 1998.

[4] T. Dierks and C. Allen. The TLS protocol version 1.0. IETF RFC 2246, January 1999.

[5] D. Ferraiolo and D. Kuhn. Role based access control. In *15th National Computer Security Conference*, Washington, D.C., October 1992.

[6] S. Godik et al. eXtensible Access Control Markup Language Version 1.0. http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf, February 2003.

[7] A. Herzberg, Y. Mass, J. Michaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, Berkeley, California, May 2000.

[8] A. Hess. Content-triggered trust negotiation. Master's thesis, Brigham Young University, February 2003.

[9] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. Seamons, and B. Smith. Advanced client/server authentication in TLS. In *Network and Distributed System Security Symposium*, San Diego, California, February 2002.

[10] R. Jarvis. Selective disclosure of credential content during trust negotiation. Master's thesis, Brigham Young University, April 2003.

[11] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall PTR, 2002.

[12] J. Kohl and C. Neuman. The kerberos network authentication service (v5). IETF RFC 1510, September 1993.

[13] N. Li, J. Mitchell, Y. Qiu, W. Winsborough, K. Seamons, M. Halcrow, and J. Jacobson. RTML: A role-based trust-management markup language. http://theory.stanford.edu/˜ninghui/papers/rtml.pdf, 2002.

[14] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a "policy neutral" access control policy. In *New Security Paradigm Workshop*, Lake Arrowhead, California, September 1996.

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

[16] K. Seamons, M. Winslett, and T. Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *Network and Distributed System Security Symposium*, San Diego, California, February 2001.

[17] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *3rd Interna-*

*tional Workshop on Policies for Distributed Systems and Networks*, Monterey, California, June 2002.

[18] M. Wilikens, S. Feriti, A. Sanna, and M. Masera. A context-related authorization and access control method based on RBAC: A case study from the health care domain. In *Symposium on Access Control Models and Technologies*, Monterey, California, June 2002.

[19] W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, Hilton Head, South Carolina, January 2000.

[20] M. Winslett, T. Yu, K. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. Negotiating trust on the web. *IEEE Internet Computing*, 6(6), November/December 2002.

[21] F. Yergeau. UTF-8, a transformation format of ISO 10646. IETF RFC 2279, January 1998.

[22] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH authentication protocol. I-D draft-ietf-secsh-userauth-16.txt, September 2002.

[23] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH connection protocol. I-D draft-ietf-secsh-connect-16.txt, September 2002.

[24] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. I-D draft-ietf-secsh-architecture-13.txt, September 2002.

[25] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH transport layer protocol. I-D draft-ietf-secsh-transport-15.txt, September 2002.

[26] T. Yu, M. Winslett, and K. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), February 2003.