2-20-2003

Flexible and Scalable Public Key Security for SSH

Yasir Ali

Dartmouth College

Adding Public Key Security to SSH

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Yasir Ali

DARTMOUTH COLLEGE

Hanover, New Hampshire

Feb, 20th, 2003

Examining Committee:

Sean Smith (chair)

Edward Feustel

Christopher Hawblitzel

Carol Folt

Dean of Graduate Studies

Abstract

SSH, the Secure Shell, is a popular software-based approach to network security. It is a protocol that allows user to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another. It provides authentication and encrypted communications over unsecured channels. However, SSH protocol has an inherent security flaw. It is vulnerable to the “man-in-the-middle Attack”, when a user establishes his first SSH connection from a particular client to a remote machine. My thesis entails designing, evaluating and prototyping a public key infrastructure which can be used with the SSH2 protocol, in an academic setting, thus eliminating this vulnerability due to the man in the middle attack. The approach presented is different from the one that is based on the deployment of a Certificate Authority. My scheme does not necessarily require third party verification using a Certificate Authority; it is decentralized in nature and is relatively easy to set up.

Keywords used: SSH, PKI, digital certificates, Certificate Authority, certification path, LDAP servers, Certificate Revocation List, X509v3 Certificate, OpenSSL, mutual authentication, and tunneled authentication.

Acknowledgments

I want to thank Professor Sean Smith for his guidance, assistance and unremitting support over the last two years. He has always been always there to guide me and provide his insight on all the technical and non-technical aspects of this thesis work. His intellectual creativity, and insightful suggestions have been invaluable for my thesis research. I could never thank him enough for being an excellent mentor and a wonderful person. Sometimes words are just not enough to express one's heartfelt gratitude.

My thanks also go to the other members of my thesis committee – Professor Chris Hawblitzel, who assisted me in setting up and using Java Cryptographic API for my project; Professor Edward Feustel for the discussions during the course of the project, which helped me clarify my concepts.

I would like to thank Professor Fillia Makedon for guiding and assisting me in every possible way throughout my course of study. I would also like to thank Carl Ellison for his discerning suggestions that were incorporated in developing a solution that is proposed in this paper.

This work was supported in part by the Mellon Foundation, Internet2/AT&T, and by the U.S. Department of Justice, contract 2000-DT-CX-K001. Therefore, special thanks to them for making this project possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This research thesis develops a *minimal* and yet *scalable* solution for the "Man-in-the-Middle-Attack" problem in the SSH protocol.

This paper offers guidelines to develop a practicable and a viable Infrastructure for certificate usage for SSH so that the user can be certain that he has *mutually*[1] authenticated with the server, in an academic setting such as the one at Dartmouth. Design constraints on the certificate authority, such as certificate generation, distribution, verification, revocation, and management have also been evaluated so that the user has to make a minimal effort to set up an ssh client and use it securely.

The deployment constraints, design considerations and scalability issues with in a Dartmouth scenario are also evaluated in this paper. Dartmouth College has a decentralized network with several LANS each independent of the other and connected to a central backbone. Each building has its own LAN that is independently managed. The network at Dartmouth has roughly 8,000 ports.

The design and code that was written to develop the prototype is open source and modifies the SSH protocol. It is based on the original open source code for TeraTerm SSH client that was written by Robert O'Callahan and OpenSSL libraries[2]. There are two ssh protocols, *the SSH1* protocol and *the SSH2* protocol. The SSH1 and the SSH2 protocols encrypt at different parts of the packets. The SSH2 protocol is a complete rewrite of the SSH1 protocol, and it does not use the same networking implementation that the SSH1 does [14.ii]. They have slightly different Key Exchange Algorithms.

[1] The user is authenticated at the server and the server key is verified on the client machine.
[2] The source code written and modified for this project is placed at www.cs.dartmouth.edu/~yasir/pki

Generally, the SSH2 protocol is considered more secure, as it avoids known vulnerabilities in the SSH1 implementation. Amongst the SSH products, there are OpenSSH and SSH clients and servers. Both the products implement the SSH1 and the SSH2 protocols. OpenSSH client and server are open source and are distributed freely with Red Hat Linux distributions. SSH client and server are developed by SSH Inc. and are available for Windows and Linux platforms.

Based on this research, Professor Sean Smith and I have also submitted a paper, “Flexible and Scalable Public Key Security for SSH”, for the second annual PKI Research Workshop.

1A. This Paper:

The next section provides the background knowledge to understand the problem. The third section discusses the possible solutions and their weaknesses as I lay groundwork for my solution and then describe my solution. The fourth section presents the design considerations for a viable Public Key Infrastructure for SSH in a “Dartmouth” setting. The fifth section presents architectural and implementation details. The sixth section highlights related work on this problem. The seventh section summarizes and concludes this paper. The last section provides definitions for the PKI terms used in this paper.

2. BACKGROUND

Putting it simply, the SSH2 protocol allows two hosts to construct a secure channel for data communication using *DSA* (public key authentication) and a *diffie-hellman-group1-*

sha1 exchange[3]. The Diffie-Hellman key exchange provides a shared secret key that cannot be determined by either party alone. The shared secret key established is used as a *session* key. Once an encrypted tunnel is created using this key, the context for negotiated compression algorithm, and encryption algorithm are initialized. These algorithms may use independent keys in each direction. The first session key established is randomly unique for every session. The OpenSSH clients, using the SSH2 protocol, allow re-keying (generating a new session key for an existing session) if the user requests during a session (by typing ~R). The SSH server provided by OpenSSH can be configured to provide automatic re-keying after a specified interval.

At this point, it is important to distinguish the difference between mutual authentication and client authentication. Mutual authentication is where two entities authenticate themselves to each other. The user on client machine verifies that the server machine is in fact the one it claims to be, similarly the server verifies that the user at the client machine is the one who has an account on the server. Client Authentication is where only the user on the client machine authenticates himself to the server. SSH supports both types of authentication. In case of client authentication, if the *fingerprint*[4] sent by the server does not match the one that is already stored in a file on the client machine, the user on the client machine would either receive a warning with the option of accepting the new fingerprint as it is, or the client would drop the connection.

It is important to realize that when a user on a client machine tries to establish a secure channel with a remote machine using the SSH2 protocol, at least four entities are involved:

---

[3] The "diffie-hellman-group-exchange-sha1" method specifies Diffie-Hellman Group and Key Exchange with SHA-1 as HASH. It is used to establish a shared key in SSH2. The SSH1 protocol uses RSA-style exchange to establish a shared key, which uses an ephemeral server key other than the server host key.

[4] Fingerprints are typically 1024 bits, rsa or dsa, public keys for the server. Public key lengths can be 512 or 2048 as well. The fingerprints are stored unencrypted on

- The SSH client on the client machine,
- The SSH Daemon (SSH server) running on the remote server host,
- The user on the client machine and
- The administrator on the remote server host.

The user can use various client machines to log in to a specific server. He may or may not be required to log on to the client machines. The role of the administrator is restricted as he is mainly responsible for configuring the security parameters on an sshd server and ascertaining that the sshd server is running properly.

There are three main parts of the SSH protocol [3]:

- Algorithm Negotiation
- Authentication
- Data Encryption

Algorithm Negotiation is mainly responsible for determining the encryption algorithms, compression algorithms and the authentication methods supported and to be used between the client and the server.

Algorithm Negotiation is followed by Authentication. Authentication is further broken down in two pieces:

- Key exchange (transport layer) [14.i,14.ii]
- User authentication (user authentication layer) [14.iii]

The purpose of the key exchange is dual. Firstly, it attempts to authenticate the server to the client. Secondly, a shared key is established which is used as a session key to encrypt all the data being transferred between the two machines. The session key encrypts the payload and a hash generated for *integrity* checking of the payload using the private key of the server. The client verifies the server's public key, verifies the server signature received and then continues with user authentication. User authentication methods that are supported and are a part of the SSH2 protocol include passwords, public key, OpenPGP certificates, X509v3 certificates, PAM, and Kerberos [16]. Currently, the latest openssh3.4 client does not provide any code that verifies certificates or certificate chains. Certificates are merely treated as public keys. If a key blob, which is the technical name for a data structure that loads the public key, contains a certificate instead of a public key, openssh3.4 has routines that can extract the public key out of the certificate and after that it only uses that public key for authentication and integrity checking purposes. Once Authentication is successful, one of the negotiated encryption algorithms is used to encrypt the data transferred between the two machines. Other features that are a part of ssh clients include port forwarding, however such features will not be discussed in this paper.

Following is a diagram [14.ii] showing the key exchange mechanism that comes after the "algorithm negotiation" stage. The key exchange produces two values: a shared secret ***K***, and an exchange hash ***H***. Given the following variables,

n is a number of bits the client requested,

p is a large safe prime,

g is a generator for a subgroup of ***GF(p)***, usually set to be equal to 2,

q is the order of the subgroup;

V_S is Server's version string;

V_C is Client's version string;

K_S is Server's public host key;

I_C is Client's **KEXINIT** message and

I_S is Server's **KEXINIT** message (which have been exchanged before the key exchange begins).

The client generates a random number ***x*** where ***(1 < x < q)*** and the server generates a random number ***y*** where ***(0 < y < q)*** and initiate the protocol as shown below [14.i,14.ii]:

Figure 1 Transport Layer Key Exchange

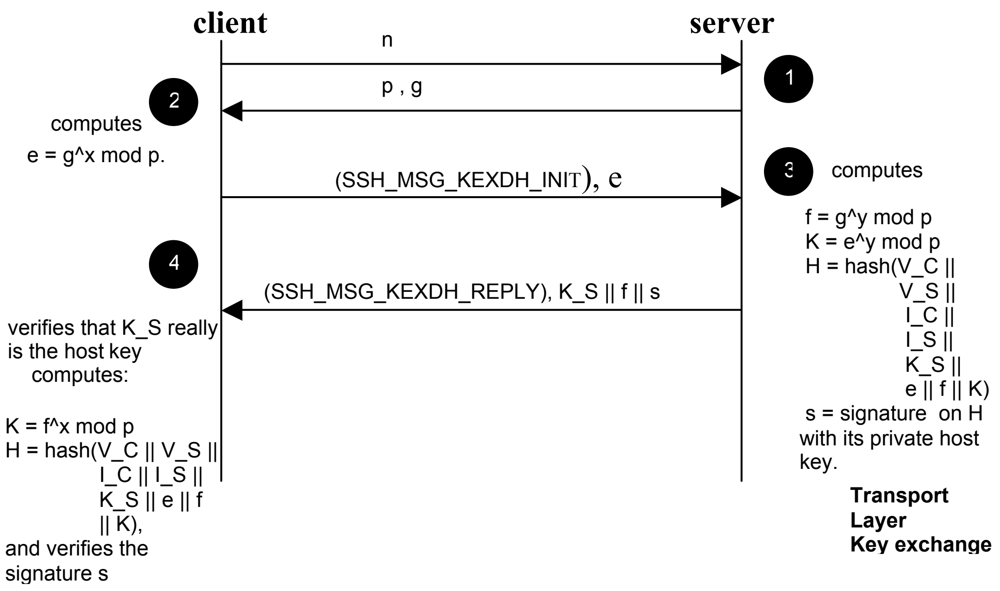
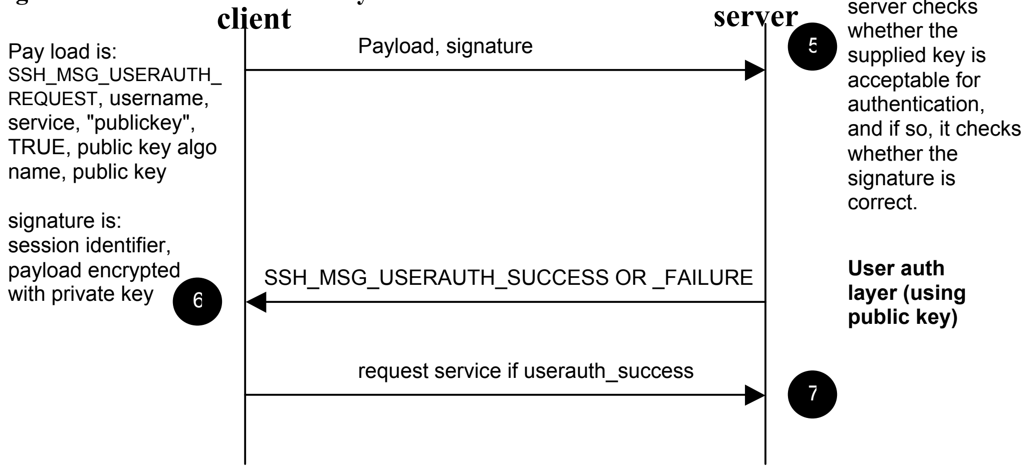
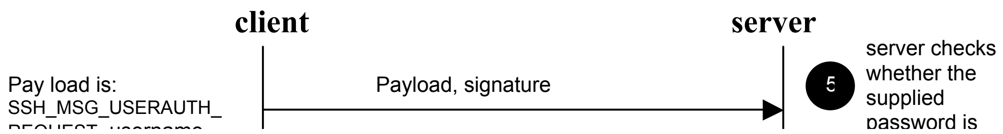

client
server
n
1
p , g
2
computes
e = g^x mod p.
(SSH_MSG_KEXDH_INIT), e
3
computes
f = g^y mod p
K = e^y mod p
H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K)
s = signature on H with its private host key.
4
(SSH_MSG_KEXDH_REPLY), K_S || f || s
verifies that K_S really is the host key
computes:
K = f^x mod p
H = hash(V_C || V_S || I_C || I_S || K_S || e || f || K),
and verifies the signature s
Transport Layer Key exchange

Figure 2 User Authentication Layer


client
server
Pay load is: SSH_MSG_USERAUTH_REQUEST, username, service, "publickey", TRUE, public key algo name, public key
signature is: session identifier, payload encrypted with private key
Payload, signature
5
server checks whether the supplied key is acceptable for authentication, and if so, it checks whether the signature is correct.
SSH_MSG_USERAUTH_SUCCESS OR _FAILURE
6
User auth layer (using public key)
request service if userauth_success
7


client
server
Pay load is: SSH_MSG_USERAUTH_
Payload, signature
5
server checks whether the supplied password is

The OpenSSH client, which is an open source ssh client using the SSH2 protocol, manages the key pairs as follows [16]:

- Each "user" creates a public/ private key pair, if he intends to use "public key authentication" on any client machine. However, that public key fingerprint needs to be added in to the database of the server, before authentication can proceed.

- Similarly the sshd (on the server) maintains private and public key pairs created by the root under etc/ssh. Typically there is a key pair based on *RSA* and another key pair based on *DSA*.

- The user account on the client machine maintains a database of all the public keys of the ssh servers to which a user logged in using SSH ($HOME/.SSH2/knownhosts)

- If the client does not have a **matching** public key of the server in $HOME/.SSH2/authorization, the user can configure the security on his machine so that it accepts the public key provided by the remote server and overwrites the existing one. It prompts the user to save the updated public host key. However the user can configure the security, such that in such a scenario, the connection would be declined.

- If the client does not have a public key (at all) of the server machine (i.e. it is connecting to the remote machine for the first time through that client machine), it accepts the fingerprint provided by the remote machine and stores it locally under that specific user account.

Given this brief description of how the public keys are managed, it is easy to deduce that the client blindly trusts the server and accepts its public key during a “first time” connection. An intruder or an attacker can intercept in such an exchange scenario required to establish future secure sessions and render the SSH channel to be insecure. This brings us to a point where we can evaluate the various security vulnerabilities in ssh.

2A. The Man in the middle Attack

Suppose Alice (user) wants to talk to Bob (server) and a malicious user Trudy wants to play the man-in-the-middle attack.

- ***Alice*** initiates a connection with ***Bob***.
- ***Bob*** sends his public key to ***Alice***, which ***Trudy*** intercepts;
- ***Trudy*** then sends her own public key to ***Alice***.
- ***Alice*** accepts the new public key and stores in its database. If it was the “first time authentication”, it would blindly add the public key provided by ***Trudy*** thinking that it is ***Bob***’s public key.
- ***Alice*** then sends her username and password to ***Bob*** that is again intercepted by ***Trudy.***

- ***Trudy*** decrypts Alice's username and password using the session key and her private key,
- ***Trudy*** then encrypts Alice's credentials using the public key provided by ***Bob*** and forwards the new packet to him.
- ***Bob*** authenticates ***Trudy*** thinking that it authenticated ***Alice***.

Trudy can now be really nasty, if she sends ***rm –rf**** command to Bob and Bob deletes Alice's user account. Trudy can also just accumulate user id and password pairs. She might just pass Alice on to Bob, so Alice's session works as normal. Except Trudy can log in later.

2B. Spoofing Attack

Spoofing is when user ***A*** claims to be user ***B*** and establishes a secure channel with host ***C***. ***C*** thinks that it has established a secure connection with ***B*** but in fact it has established a channel with ***A***. User ***A*** hides his real identity and forges a false identification. Such an attack is possible as we do not trust the underlying network.

User spoofing is possible when an innocent user ***B*** on a client machine attempts to establish a connection with a remote host ***C***. A malicious server ***A*** intercepts the channel when it is being initiated, fakes to be remote host ***C*** and replies back with its own public key.

If the SSH client configuration on ***B*** is set to non-strict host key checking (which is the default installation configuration), it would ask the user to overwrite the previous key stored in its database for host ***C*** and proceed with establishing a connection. If it was the "first time" authentication by ***B*** on that specific client, he would accept the server host key anyway. In a common scenario, the user performs password authentication to the remote server. In that case, the malicious remote server can accept the credentials provided by B and then output the error message that invalid password was provided (the user can re-enter his password twice and the server would give the same error message) and then disconnect. For any future sessions, an attacker can now use B's credentials retrieved from malicious server ***A*** and then pose to be user ***B***.

The "Spoofing Attack " stems from the same basic vulnerability where a traveling user has to retrieve the server public key for the first as was the case in the "Man in the Middle Attack".

2C. A lesser significant Knownhosts2 file Attack

Suppose a hacker hacks Alice's password on a specific server machine. He can then proceed with his attack on other machines by looking up Knownhosts2 file in Alice's home directory (~alice/.SSH2). He can attempt to break into all the listed machines in the Knownhosts2 file using Alice's username and password. If Alice used the same username and password for those machines, in all likelihood, the hacker would successfully break into those machines as well.

2D. SSH Security Vulnerability

Given this description of "spoofing" and "Man in the middle Attack", we now focus on the reasons why SSH authentication model is weak.

1. When the server sends its public key and its signature, since they are not signed by a third party, it is trivial for an attacker to sit in the middle and intercept the connection, and send *his* own public key and signature instead;

2. When the user sends his own password, to the attacker thinking that he is the server, the username and password are compromised by the attacker as shown in the previous attack scenarios.

3. If it is the first time the user is connecting to a host and hence does not have the server's public key stored locally, he will be none the wiser. The attacker can send his public key to be stored instead of the real server. If the user does have the server's public key, when the attacker sends his own public key instead, the user will generally receive a warning like “Warning: server's key has changed. Continue?” Most users typically hit “Yes” and do not realize the risk that someone might be trying to compromise the security of his connection by putting the attackers own public key in use instead of the real server’s public key.

Unless both parties have some *pre-established* shared data, any authentication system will probably be susceptible to a man-in-the-middle attack.

3. POSSIBLE SOLUTIONS

Before we delve into discussing our solution, let us explore existing authentication methodologies that can be used or integrated into SSH Clients to resolve the problems identified above. The prerequisites and the drawbacks for each one of those methodologies would also be discussed. After analyzing these authentication solutions, our proposed authentication solution would be presented and evaluated. At this point it is pertinent to mention that the latest SSH Clients by SSH Inc., provide a solution to resolve the vulnerabilities discussed in this paper [16i]. Their solution is based on the deployment of a Certificate Authority that issues certificates to all the SSH servers. The SSH client retrieves the CA Certificate first and then uses it to perform server certificate verification. Their solution is discussed in greater detail, in the “related work” section of this paper.

3A. Solution 1: TLS (Transport Layer Security using User Certificates)

TLS is based on Secure Socket Layer protocol (SSLv3), and performs the SSL handshake for authentication. It allows for mutual authentication between the user and the server, using certificates.

The user on the client machine must have:

- User Certificate.
- *CA Certification Path*[5] installed on the client machine.

The server must have:

- Server Certificate
- CA Certification Path (or the CA Certificate)

By using certificates, the problem caused by the lack of proper verification of the fingerprints in the SSH2 protocol would be solved, as now when a user receives a new server certificate (instead of a new fingerprint), the user can verify whether the root CA signed the new certificate. If so, then it is a valid certificate and it can continue with authentication. It would no longer be a judgment call, where the user decides on his own

[5] A certification path is an ordered list of certificates. If a certification path meets certain validation rules, it may be used to securely establish the mapping of a public key to a subject. Typically , the last node in the ordered list of certificates is the CA certificate. A CA certification path is a list of certificates that establish a validation path for a user certificate to the CA certificate

will if he wants to accept the fingerprint or not. Similarly when it is a first time authentication, it can verify that the certificate received is actually the right one.

Prerequisites:

A successful deployment of a Root Certificate Authority is integral to the success of this solution. A Certificate Authority is responsible for generating, revoking, signing, storing and managing certificates. A 'Trusted' Root certificate authority is a certificate authority whose signing of a certificate serves as a crucial link for it to be authentic to the connecting hosts. An 'Intermediate' certificate authority is a certificate authority that vouches for the authenticity of other certificate authorities and it has a root certificate authority vouching for its own authenticity. However, it does not issue any user certificates.

OpenSSL provides a set of open source libraries that allow verification of certificates and certificate chains that is needed for such a solution. Currently, this library is being used in several commercial products. RSA's B-safe libraries also provide a similar set of libraries.

Microsoft Certificate Store (viewed from inside Internet Explorer or Microsoft Management Console in Windows 2000 and XP) contains a list of root trusted CA, that are used in verification during SSL handshakes. Using OpenSSL libraries and the Microsoft crypto libraries, certificates can be accessed and verified from the Microsoft Certificate Store. This lays out a clean design for setting up, storing and accessing underlying user certificate structures. Netscape and Mozilla also offer browser-based certificate stores that can be accessed in a similar fashion.

In terms of ease of use, most Certificate Authorities that run on Windows Operating System are easy to set up and use. *Microsoft Certificate Authority* runs on a Windows 2000 Advanced Server. There are also open source CA's such as OpenCA that can be downloaded and set up.

Dissemination of user certificates in the Dartmouth scenario would require a web interface that would allow users to download their user certificates and certification paths. The web interface would be protected by a Kerberos log in mechanism that is already deployed fully at Dartmouth. The user credentials could be verified via Kerberos log in or using default domain log in mechanism provided by Windows 2000. Bob Brentrup is currently heading the deployment of a PKI and Trust Hierarchies at Dartmouth.

Drawbacks*:*

When a user uses a new client machine that is outside the realm of Dartmouth, he would not have the Certification Path installed on the machine, therefore he will not be able to verify whether the certificate received by the server is valid or not. Suppose that the Dartmouth Certificate Authority protected by a login mechanism is accessible from outside the Dartmouth realm[6], the user successfully retrieves the CA certificate. However, he still needs to verify that he received the correct certificate. An attacker can use IP spoofing or DNS spoofing to send him a fake CA certificate. The existing solutions for this problem require setting up CA hierarchies that lead to scalability issues. The TLS solution would work if the Dartmouth CA was certified by one of the standard browser trust roots which is an expensive proposition.

[6] this poses security risks as hackers can attempt to access user certificates by password guessing. According to some studies, students tend to have passwords that can be guessed by dictionary attacks.

Secondly, user certificates installed on a client machine, no longer remain user specific. Instead they become machine specific. Therefore if a user downloads his certificate on a specific machine, he will have to delete it, before he leaves the machine. Otherwise a malicious user can impersonate him by using his certificate on that machine. This problem would come up on machines where the users are not required to log in to use the machine[7].

Thirdly, most users want to use “hassle” free authentication mechanisms, where once the machine is configured for authentication, they will not have to modify it again. This solution requires vigilance on the part of the user.

Fourthly, dedicated resources, both human and machine, are needed to manage and track the certificates issued to users, which makes it an expensive scheme. Furthermore it poses security risks as anyone with a Kerberos log in can access the Certificate Authority, including students who like to consider themselves hackers.

3B. Solution 2: Smart Cards/Tokens (using X509 Certificates)

Smart cards can store certificates on them. The certificate image and the certificate path are stored on the local storage on the smart cards may also be protected by a pin number/pass-phrase if they store the private key corresponding to the certificate.

A smart card is plugged into a smart card device, which is plugged into the USB port. It may export its x509v3 certificate image and certification path onto the Microsoft certificate store. The Certificate image must contain the public key and it can also have a private key that would either reside on the smart card or in the certificate manager. The user can

[7] For instance machines with Windows 98, ME, or Irix (sgi) with guest accounts

then use his ssh client to perform TLS authentication using smart card certificates. Ideally when the user unplugs the smart card, it should remove the certificate from the Microsoft certificate store, however not all the smart cards work that way. *USB-Token*s can be directly plugged into a USB port and do not require a smart card reader. Unlike the traditional smart cards, they cannot be kept in a wallet, but can be put on a key chain.

It solves the problem where if the user used a machine outside the realm of Dartmouth, he did not have access to the certificate. Given he has access to a smart card for traditional smart card readers, and the right drivers installed for USB-token he can use the ssh client from anywhere.

Drawbacks:

Smart card is a hardware solution where the user must carry this piece of equipment with him. In a Dartmouth scenario, students would be required to carry these cards with them. Unless these smart cards become a part of the Dartmouth ID card, there is a low feasibility for a successful deployment of smart cards. Secondly, smart card reader devices need to be installed on every machine that would be used for ssh. For USB-tokens, one would need to install drivers on all client machines. Furthermore a fully developed Public Key Infrastructure would still be needed where a certificate authority would issue smart card certificates; the system administrator would burn the certificate image onto the smart card, and keep track of those certificates and the smart cards as well. Lastly, if the user goes outside of the Dartmouth realm, there is no certainty that he would find the same vendor specific smart card reader on a machine, where he can plug in his smart card and use ssh. As yet smart card technology is not scalable and is expensive. Furthermore there are various smart card vendors in the industry and are not fully interoperable with each other. PKCS #11 provides a method for accessing hardware devices in a device neutral manner, however not all the smart card devices implement that yet. USB-tokens need driver and software installation on a client machine before they can be used. Typically these installations are also vendor specific.

3C. Solution 3: TTLS (Tunneled Transport Layer Security using Passwords)

This is an extension to TLS, where the user is not required to have a client certificate; instead a user name and a password are used.

- The Server sends the server certificate.
- The Client verifies the server certificate by checking if a trusted root authority signed it.
- If verification succeeds, the user sends his username and password over a channel encrypted by a session key. There are various protocols that can be used for this purpose. Some of them are one-time password protocols, challenge response protocols, or even sending the plaintext username and password through the encrypted channel.

This method of authentication is similar to the SSH2 protocol where the user verifies the server fingerprint by looking up its database. If the fingerprint matches, it creates an encrypted tunnel (using symmetric session keys) and then the user sends his username and password. TTLS is superior to the SSH2 protocol mainly because of its server certificate verification mechanism where the signature of a root CA on the server certificate is verified.

This authentication methodology has the advantage that it is easily deployable in an academic setting or within a corporate set up, as it does not require the deployment of user certificates.

However the drawback is the same that the user needs to have the CA certification path (or the CA certificate) installed on the client machine, which means that he needs to somehow access it securely and install it on his machine if it is not already installed. *PEAP* (Protected Extensible Authentication Protocol) is a similar authentication protocol used in wireless networks, using the same underlying concepts as that of TTLS. Therefore this paper will not discuss PEAP.

3D. Solution 4: LEAP (Lightweight Extensible Authentication Protocol)

This proprietary authentication method is developed by CISCO. It provides mutual authentication between the user on a client machine (*supplicant*), and a radius server (*authentication server*), through a wireless Access Point (*authenticator*) on a wireless network. What makes this authentication protocol so interesting is that it does not use server certificate or a public key to authenticate the server. Instead, it uses a two way Challenge Response mechanism to validate the radius server to the supplicant. Following is a quotation from Cisco's website[8]:

> *"For its Aironet solution, Cisco created an authentication scheme based on the Extensible Authentication Protocol (EAP) called EAP-Cisco Wireless or LEAP. Using the 802.1x draft standard for port-based security as a foundation, but with the necessary modifications for WLANs, LEAP provides mutual authentication between Cisco Aironet client cards and the backend Remote Authentication Dial-In User Service (RADIUS) server"*

[8] http://www.cisco.com/warp/public/784/packet/jul01/p74-cover.html

Wireless Security Research group at the MISSL Lab at University of Maryland analyzed LEAP[9] and deduced that LEAP although offers mutual authentication, poses security risks in its authentication mechanism. A shortcoming of LEAP is the fact that when authentication takes place it has to take place in the "Open mode”. If a client is trying to authenticate for the first time, he needs to know the current link layer communication (*WEP*) key. Their work shows that authentication in open mode is very insecure and vulnerable.

On the following page is a table summarizing the differences between these authentication schemes.

[9] http://www.missl.cs.umd.edu/wireless/ethereal/leap.txt

Table 1 Authentication Schemes

Authentication Method	**TLSv1**	**TTLS**	**LEAP**	**SSH**
Provides Mutual Authentication	Yes	Yes	Yes, but is vulnerable	Yes, but is vulnerable
Provide Client Authentication	Yes	Yes	Yes	Yes
Protocol Summary	Establish TLS session and validate certificate at both ends.	Establish TLS session, client validates server certificate, server validates username and password	Technical Specifications unavailable. Sniffing over connection shows a mutual challenge response mechanism based on hashing.	Establish an SSH session, validate server public key from database, and clear text username and password sent through encrypted channel
Server Certificate / public key	Certificate required	Certificate required	Not used	Server public key required
Server Certificate / public validation	Using CA certificate path or OSCP	Using CA certificate path or OSCP	Not used	Server public key verified if stored in database
Client Certificate / public key	Required	Can be used, not required though	Not used	Can be used, not required though
Client Certificate validation	Using CA certificate path or OSCP	Using CA certificate path or OSCP	Not used	Public key verified if stored in database
Username and Password	Not used	Can be used, not required though	Required	Commonly used
Password validation	Not needed	Password validated in encrypted channel using, MSCHAP, PAP, MSCHAPv2, CHAP, none (clear-text)	Password validation seems to use MSCHAPv2	Clear-text password validation in encrypted channel
Allow Secure mobility to users	**No, as certificates needed**	**No, as CA cert needed**	**No, as it is "port access" based authentication**	**No, as server key verification needed**
Provides Encrypted Channel after server validation	Yes	Yes	Yes	Yes
Drawbacks:	- Too many certificates to manage. - Difficult Deployment - Mobile user cannot trust	- Mobile user cannot trust CA Certification path retrieved	- Specifications not available - Analysis, by MISSL Lab show vulnerabilities in mutual	- Open to Man in the Middle Attack - IP Spoofing - Mobile user cannot trust

	CA Cert retrieved		authentication scheme	server public key

3E. Our Solution:

So far we have seen that none of the previous solutions are “lightweight”. Also, they do not fully resolve the following two issues:

- They do not make “first time” authentication on a new client machine secure for a traveling user.
- They are not easy to deploy in an academic setting where many users share a group of machines, or use their own machines.

Furthermore, it has been observed that most of the users are comfortable with using usernames and passwords instead of using client certificates, which is a hassle for both,

- the user as he has to manage it on his machine and is required to install it on any machine he may intend to use for ssh, and
- the system administrators who may be responsible for issuing, or revoking certificates and managing the certificate authority and a registration authority.

Lastly, system administrators desire minimal maintenance once they have set up the ssh server. The desired goals of our solution are as follows[15]:

- The solution should enable users from borrowed (but trusted) clients to establish trusted connections to their home machines.
- The solution should be adoptable in the near-term by small groups of users with only a small delta from the current infrastructure.

- The solution should accommodate users in domains where conscientious sysadmins can set up trustable and usable CA services.
- The solution should accommodate users in domains where no such services exist.
- The solution should *not* require that a new universal PKI hierarchy be established before any of this works.
- The solution should *not* require that a user memorize the fingerprints of all servers he wishes to interact with.

To solve the problem, the client needs to have some trust root upon which to build the conclusion that the binding of the server's public key to identity is meaningful. The "small delta" and "no new universal PKI" constraints mean that we cannot hard-code this trust root into the client (and that different users might have different roots). The "no memorization" constraint means that the user cannot bring it with them.[15]

This analysis thus forces us to have the client download the user's trust root over the network. Since changing how the SSH protocol itself works would also violate the "small delta" constraint, we need download this data out of band. However, this raises a conundrum: if the user cannot trust the network against man-in-the-middle attacks on the public key the server sends in SSH, how can the user trust the network against man-in-the-middle attacks against this out of band data? [15]

To answer this, we use a *keyed MAC*—a "poor man's digital signature." Also known as a keyed hash, a keyed MAC algorithm takes a message M and a secret key k, and produces a mac value Mac(M; k) with the property that it is believed infeasible to find another M_0; k_0 pair that generate the same keyed MAC. Thus, if Bob knows secret key k, and retrieves a message M accompanied by a MAC value h which he confirms as

Mac(M; k), then Bob can conclude that M was produced by a party that knew k and has not been altered in transit. Our constraints dictate that we cannot force the user to memorize a public key—but users easily memorize URLs and pass phrases.[15]

Taking into consideration these design constraints, this paper presents two authentication mechanisms.

- A Centralized Minimal PKI Approach
- A Decentralized Non-CA Approach

Before we describe how these schemes work, it is pertinent to provide a brief description of a *Message Authentication Code,* HMAC, used in these approaches.

i. HMAC

HMAC [6] is a Keyed Message Authentication Code Algorithm. It requires a cryptographic hash function, H that typically uses MD5 or SHA-1 and a secret key K. The cryptographic hash function hashes the data by iterating a basic compression function on blocks of data. The byte-length (B) of such blocks is typically set to 64 bytes. The byte-length of hash outputs is 16 bytes for MD5 and 20 bytes for SHA1. The authentication key K can be of any length up to the block length of the hash function. In any case the minimal recommended length for the secret key is equal to the hash output length.

Two fixed and different strings ipad and opad are defined as follows:
('i' and 'o' are mnemonics for inner and outer):
ipad = the byte 0x36 repeated B times
opad = the byte 0x5C repeated B times.

To compute HMAC over the data `text', the following transformation is performed:

H(K XOR opad, H(K XOR ipad, text))

That is to say,

1. Append zeros to the end of K to create a B byte string (e.g., if K is of length 20 bytes and B=64, then K will be appended with 44 zero bytes 0x00),
2. XOR (bitwise exclusive-OR) the B byte string computed in step 1 with ipad,
3. Append the stream of data to the B byte string resulting from step 2,
4. Apply H to the stream generated in step 3,
5. XOR (bitwise exclusive-OR) the B byte string computed in step 1 with opad,
6. Append the H result from step 4 to the B byte string resulting from step 5
7. Apply H to the stream generated in step 6 and output the result

The key for HMAC can be of any length (keys longer than B bytes are first hashed using H). However, less than L bytes is strongly discouraged, as it would decrease the security strength of the function. Keys longer than L bytes are acceptable but the extra length would not significantly increase the function strength.

HMAC can generate a hash value for a public key that requires the user password as an input. If the user has this hash value, he can verify a public key by hashing it using his password as an input and if the hash value generated is the same as the one he already had, he can be certain that the sever key has not been tampered with. Using this simple scheme he can make sure that he has received the correct server key.

ii. Centralized Minimal PKI Approach:

This approach is similar to the solution offered by other commercial SSH vendors that deploy PKI infrastucture for their clients. It mainly differs in the following ways:

- The scheme for distributing CA certificates or CA certification paths to the user is different. In conventional solutions a CA certificate can either be installed from a file off of a floppy (if it is stored locally on the disk in a PKCS#7 file format), or through certificate enrollment mechanisms. In our approach the trusted certificates are installed in the Microsoft certificate store only, primarily because Internet Explorer is pre-installed on every other machine and they are received from an ldap server with an HMAC.
- In our approach, users are not issued certificates. Only the server is issued certificate. Protecting a Certificate Authority by a log in mechanism and allowing students to access it within and outside of Dartmouth realm makes the Certificate Authority vulnerable to attacks. Therefore, in this approach, Certificate Authority has access limited only to system administrators unlike SSH clients by SSH Inc, where users are required to either have certificates or key pairs.
- An additional component is added to the protocol that stores CA certification paths.

This approach requires that:

1. The user on the client machine must have:
   - A username and a password on the server machine, and
   - He must remember the URL from where he retrieves a hash of the certification path, and the certification path itself. HMAC is

used to generate the hash for the certificate path. It takes the user password as a Key.

2. The server must have:
   - A server certificate (verified by a root certificate or a certificate chain)
3. A Single Certificate Authority that issues:
   - Server certificates for all the machines running the ssh server,
4. A URL that is connected to an *LDAP*[10] server or a database that stores a list of usernames and their corresponding hash of a certificate path.[11]
5. A Hash generation tool that populates the LDAP database.

iii. How it works:

Suppose a Professor at the CS Department in Dartmouth goes to California to attend a conference. While he is in California, he decides to do an "ssh log in" to one of the machines in Dartmouth. He initiates the connection to ab.cs.dartmouth.edu:

1. ab.cs.dartmouth.edu sends the server certificate to the client machine,
2. As the professor is using the client machine for the first time, his ssh client has no means to verify the server certificate.
3. He types in the URL address, which is connected to an LDAP server and sends his username to it.

[10] Lightweight Directory Access Protocol. Any other database can also be used.
[11] A prototype LDAP server, using OpenLDAP was set up that stored a few usernames and their corresponding hash values. A separate tool generated those hash values. Currently the process is not automated for a centralized CA Approach.

4. The LDAP looks up the username and sends back the certification path and the hashed CA certification path corresponding to that.
5. The SSH client now has
    a. The server certificate,
    b. A hash of CA certification path,
    c. The CA certification path
6. The Professor types in his passphrase, that is used as an Initialization Key (its is also referred to as shared key and preferably should be 20 bytes) and generates a hash of the certification path received. The generated hash is compared with the hash received from the LDAP server. If they match, the professor is ascertained that he received a valid certification path.
7. The client validates the server certificate using the certification path received, and he may also install it on the client if he plans to use the same client machine in future, otherwise discard the certification path.
8. Once the server certificate is verified, a symmetric shared key is established and the user sends in his username and password to the ssh server in an encrypted tunnel.
9. The ssh server verifies the username and password, and if the verification is successful, ssh server sends an SSH_MSG_USERAUTH_SUCCESS packet, and authentication succeeds.

iv. A Decentralized Non-CA Approach:

This approach offers a solution that is simple and easy to set up. The underlying theory is the same as that of a centralized CA Approach.

a. The user on the client machine must have:

        i. a username and a password on the server machine, and
        ii. he must remember the URL from where he retrieves a hash of the server certificate. The hashing algorithm that is used to generate the hash for the server certificate takes the user password (20 byte long max) that is used as a shared key in HMAC.
    b. The server must have:
        i. a key pair
    c. A URL that is connected to a web page that stores a list of hostnames and their corresponding public key hashes.
    d. A keyed-hash generation tool on the server machine.

Continuing with the same example:

1. ab.cs.dartmouth.edu sends the server public key to the client machine,
2. The professor is using the client machine for the first time, therefore the SSH client has no means to verify the server certificate.
3. He types in the URL address, to retrieve the hash of the server key for ab.cs.dartmouth.edu. The hash is saved into a file.
4. The SSH client now has
    a. The server's public key,
    b. A hash of the public key,
5. The Professor types in his password, (a shared key and preferably should be 20 bytes) and a hash of the server's public key is generated. The generated hash is compared with the hash received from the web page. If the two hashes match, the professor is ascertained that he received a valid server public key.
6. The SSH client accepts the server's public key.

7. A symmetric shared key session already established is used and the professor sends in his username and password to the ssh server in an encrypted tunnel.

A hash generation tool is needed that can be used by the user to generate hashes for the server's public key. The hash generation tool takes in as input, the path to the directory (e.g. ~alice/public_html/ssh-hashes/) and the user password. It then creates a hash of all the public keys present in /etc/ssh/ and copies each hash as a separate file into web directory. Typically there are two public keys an RSA-based key and a DSA-based key for every ssh server. Using this tool, a user can populate his web archive of hashes that he can retrieve on any client machine via a web browser and then proceed with authentication.

v. Is it really secure?

Both the approaches presented in this paper stem from the basic fact that the user can use his password as a shared secret key to create a hash. Let us critically analyze our authentication protocols to determine if this authentication mechanism is really secure. Before we proceed, it is important to analyze the structure of a certificate.

A certificate is uniquely identified by either its Subject Name, (which is a composite of Common Name, Organization Unit, Organization, Location, State, Country, Email) or one of the X509v3 extensions that can be added to any server certificate issued. X509v3 extensions may include a server name, or a DNS address. In our case, the server name ab.cs.foo.edu would be used as a unique identifier and would be entered as a Common Name. All the fields in a certificate, including the public key, are signed by the corresponding private key component for that certificate. The self-signed root certificate can then also be appended to the server certificate. This forms a 2-node certificate chain.

Typically the private key of the server is not attached on to the certificate. However, if needed, it can be attached to the certificate and protected by encrypting it with a password. Therefore any person who knows that password can extract the private key from such a certificate. Such certificates are said to have “exportable keys”. In our case we will not use certificates with exportable keys as they are open to password attacks. Given this basic structure of the certificate being used, let us evaluate all the steps of the protocol.

Centralized CA Approach:

1. ab.cs.dartmouth.edu sends the server certificate to the client machine. An attacker who has a different server certificate issued by the same certificate authority can intercept the server certificate. Suppose he wants to pose as ab.cs.dartmouth.edu. He replaces the server certificate, with his own certificate. Certificate verification would fail in such a scenario because when the ssh client looks at the uniquely identifying field of the certificate, it would not be the one it expected. The IP address x509v3 extension would not match. If the attacker forges it to be the same, he would not be able to modify his signature to match the forged IP extension, as the CA generated the signature, therefore the certificate verification would fail again. The user would verify the signature of the attacker using the CA’s public key and match it with the credentials provided which would not be the same. This establishes the fact that the attacker cannot forge a certificate, even if he has a certificate issued by the same certificate authority.

2. The user types in the URL address, which is connected to an LDAP server and sends his username to it.
    a. A malicious user can intercept, the username and send back a different hash of a ‘modified’ certification path, and a modified certification path itself. However when the user generates the hash for the ‘modified certification path’, it would not match the hash received, as he uses his password as the “hash input key”. The attacker would not know the password used by the user. The only way to crack this protocol is by cracking the password used. That can be done by dictionary attacks or by using other such techniques to guess a password. Therefore it is important that the user selects a long password and fulfills the requirements of a good password to ensure that this protocol is secure.

3. Suppose the server certificate changes. In such a scenario, an attacker could use the old invalid server certificate from a previous session and successfully pose as the server. The user would receive the old certificate, and he would retrieve the hashed CA certification path, and the CA certification path from the ldap server. As the CA’s public key has not been modified, the user would validate the invalid server certificate, and send his username and plaintext password to the attacker. There are two ways to avoid this problem. Firstly, the usage of Certificate Revocation Lists [1,7] or OSCP [10] can inform users that a certificate has been revoked. In that case, the user would check whether the certificate has been revoked or not. If it has been revoked, he would not continue authentication with an invalid certificate. This approach requires addition of components to the existing protocol, which would query the status of the certificate, retrieved at the client end and manage or store certificate revocation [2] information at the server end. A simpler solution to the problem is that server

certificates issued are irrevocable. What that means is that once a server is assigned a certificate, it cannot be changed until it expires. Once the server certificate is expired it would be recertified. This option does not complicate the protocol. This protocol is secure as long as the private key corresponding to the certificate is protected and safe. We chose the latter solution. The primary reason for that is that we want minimal extensions to the SSH2 protocol, so that it can be used the way it is used now. SSH is used as a replacement for Telnet, therefore it should be kept as simple as possible. Secondly, the security of any public key infrastructure relies on the fact that private keys are secure. If the private keys were compromised, any PKI deployment would be compromised.

Decentralized non-CA Approach:

1. ab.cs.dartmouth.edu sends the public key to the client machine. An attacker intercepts and replaces it with his own public key. Once the client receives the public key, he retrieves the keyed-hash of the public key. He hashes the public key and compares it with the hash received. The two hashes would not match.

2. The attacker now attempts to send his own keyed-hash instead of the one retrieved from the website. To be able to generate a valid hash, the attacker needs to know one valid piece of information, the passphrase that the professor used to generate the hash. Unless he gets access to that passphrase, he will not be able to generate a valid hash. Therefore if he replaces a valid hash from the website with one of his own, that hash would not match to the one that the professor would produce using his passphrase on the spoofed public key received. Therefore attackers attempt to hack the professor’s account are foiled.

3. Suppose the server public key changes. In such a scenario, the professor would be vulnerable to a replay attack. There are two ways in which such an attack can be avoided. In a simpler model, the professor can be informed via email, as soon as possible, of the change in the server key so that he can update his web page repository of hashes. This methodology is not very secure as there are security risks related to email spoofing that come into the picture. The second methodology requires further modifications to the SSH2 protocol. In the existing the SSH2 protocol, once the server is verified, the user sends his username and password as plaintext in the encrypted channel. Replacing the current user authentication method with a “challenge response” protocol would ensure that a replay attack cannot be performed. Incase the old private key was compromised; the professor would not be vulnerable to an attack. However he will not be able to connect to the server until he updates his web page repository.

4. Attacks due to DNS Spoofing are also defied by the verification of the host key. Suppose that the user logs in on a client machine for the first time and types in ab.cs.dartmouth.edu that maps to 129.172.111.4. However, an attacker spoofs it so that the user attempts to connect to 129.172.111.5 instead of 129.172.111.4. The fact that the attacker can only posses the public key of the server and not the private key, implies that he cannot generate the signatures that validate the payload during the key exchange, therefore he can not successfully establish a shared secret key which follows a successful server authentication at the transport layer. In a scenario where the physical IP address of a server changes (e.g. ab.cs.dartmouth.edu has its IP address updated to 129.172.111.6 from 129.172.111.4), the user would receive a warning that IP address has been changed. If the user continues, the host key would be verified and the protocol would proceed as usual.

To recapitulate the two authentication methodologies discussed and analyzed so far, following diagrams (figures 3 and 4) displays how the protocols would proceed incase of successful authentication, where the mobile user is on a client machine that does not have the server keys for the ssh server that he connects to for the first time.

Figure 3 Semi-centralized minimal PKI Scheme

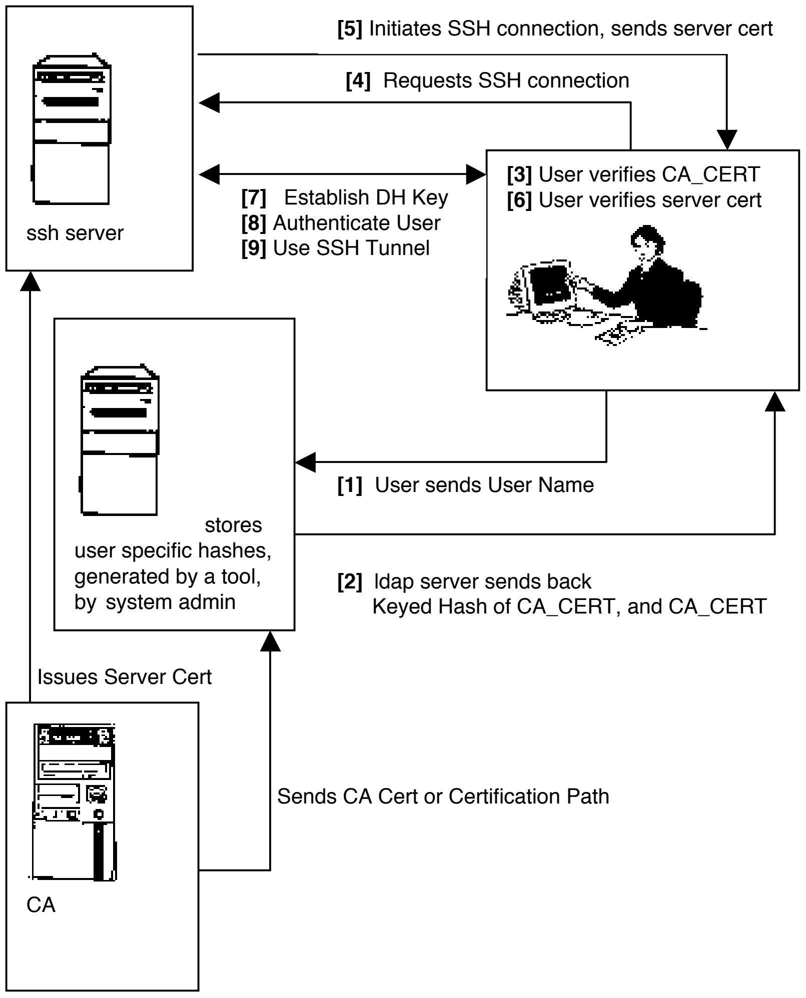
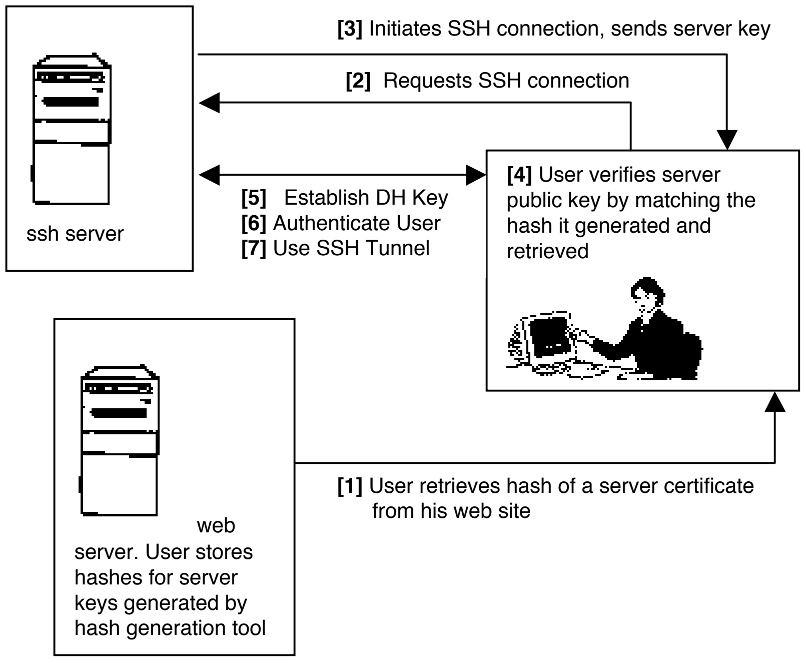

[5] Initiates SSH connection, sends server cert
[4] Requests SSH connection
[3] User verifies CA_CERT
[6] User verifies server cert
[7] Establish DH Key
[8] Authenticate User
[9] Use SSH Tunnel
ssh server
[1] User sends User Name
stores user specific hashes, generated by a tool, by system admin
[2] ldap server sends back Keyed Hash of CA_CERT, and CA_CERT
Issues Server Cert
Sends CA Cert or Certification Path
CA

Figure 4 Decentralized Scheme


[3] Initiates SSH connection, sends server key
[2] Requests SSH connection
[4] User verifies server public key by matching the hash it generated and retrieved
[5] Establish DH Key
[6] Authenticate User
[7] Use SSH Tunnel
ssh server
web server. User stores hashes for server keys generated by hash generation tool
[1] User retrieves hash of a server certificate from his web site

4. DESIGN CHOICES FOR DARTMOUTH'S ACADEMIC SETTING

One of the goals of this paper was to come up with a *deployable*, *scalable* and a *manageable basic* Public Key Infrastructure for our authentication scheme to work with ssh. The existent network infrastructure at Dartmouth is ideal to emulate a complex real world network topology. It has the state-of-art computer network technology with wireless network deployment in progress. Successful deployment in such a setting would be a leading example for the academia and the industry to follow.

In this section, several design questions are presented to the reader. The implications of the alternative solutions for each design question are evaluated and a specific solution is opted based on that.

4A. Suitable and scalable Certificate deployment methodology

How many certificate authorities should be set up for "Centralized CA Approach"?

1. One for each LAN (several LANs within each building)
2. One for each Department (roughly 20 departments)
3. One for each School (Undergraduate, Graduate, Medical and Engineering Schools)
4. Only one for the whole Dartmouth Network that can only be accessed from within Dartmouth, only by the administrators.

We opt for setting up only one Certificate Authority [8] for the whole Dartmouth network. The reasons for this choice are as follows:

First of all, this eliminates the hassle of intermediate certificates (cross certificates or bridge certificates) that would be generated if we have multiple Certificate Authorities within Dartmouth realm. We will not be required to set up trust policies between them. The system administrators will not have to co-ordinate with each other to determine the rank of their certificate authority in a Certificate Authority Hierarchy. In a complex Certificate Authority Hierarchy, compromising the security of one CA could compromise the whole PKI. Furthermore, if a system administrator decides to change the root certificate on a CA, his change would be rippled through all the other CA's who may have its certificate in their certification paths. We follow the "KISMIF" principle (Keep it simple, make it functional).

Secondly, as the Certificate Authority is the most crucial building block in a Public Key Infrastructure, it is really important that the Certificate Authority is protected from any possible security breach or unauthorized access. All effort should go in securing just "one" CA instead of several CAs.

Thirdly, it is easy to make just one CA fault tolerant, and it would require lesser resources. To achieve greater fault tolerance that Certificate Authority can be backed up every day on another machine, and if on a given day, the Certificate Authority crashes, the backup machine can quickly take its place.

Fourthly, it is logical to have multiple CA if they allow "load sharing". However in our scenario, only the machines running ssh servers are issued certificates. The users are not issued certificates. The root CA would not be required to extensively manage

Certificate Revocation Lists [1]. Certificates would only be revoked if the ssh server requests a new certificate if it is lost or deleted from the machine. The fact that there are no user certificates makes this option suitable.

Lastly, the system administrators would be expected to ensure that they have the latest CA certification path installed on their LDAP servers. They will not be required to update the CA certification paths on each ssh server. They would only update the certification path stored in the LDAP server, which would then update the hashed certificate paths for each user. That process can be automated. The LDAP server can access the hashes of the user passwords from /etc/password file. The ldap server would then generate the hashed certification paths using user specific password hashes that would be different for each user. In that case the clients would first generate the password hash and then generate the hash on the CA certification path received.

What data format should be used to generate the digital certificate?

1. X509v3 Certificates, or
2. OpenPGP Certificates

We opted for X509v3 certificates. The reasons for that are multifold. First of all, X.509 standard [4] constitutes a widely accepted basis for such an infrastructure. Secondly, Microsoft Certificate Store and OpenSSL libraries are both interoperable with x509v3 certificates. Thirdly, x509v3 certificates support extensions that can be added into a certificate, which can then uniquely identify a certificate on the basis of its IP address extension.

When should the certificate expire? Should they be revoked or not?

There are three possible choices:

1. Certificates should have short life spans.
2. Certificates should expire in one to two years.
3. Certificates should have long life spans.

Short life spans would unnecessarily burden the system administrator, who would end up upgrading the ssh server every now and then with newer server certificates and consequently updating the data stored and accessed through the LDAP server. It would require active management of Certificate revocation lists [9] as well. Considering the analysis for prevention of a replay attack, the server certificates should have long life spans.

A certificate can be revoked for the following legitimate reasons:

- The private key may have been compromised; in which case, the certificate should be treated as revoked.
- The binding in the certificate chain, $\boldsymbol{c_{0..j}}$[12],is no longer valid, in which case, $\boldsymbol{c_j}$ should be treated as invalid if it contains the same binding as $\boldsymbol{c_{0..j}}$.
- The binding may still be valid, but the issuer doesn't want to vouch for it anymore, in which case, $\boldsymbol{c_j}$ should still be valid.

However it is important to realize that certificate revocation complicates the model and opens up an opportunity for a security breach due to the replay attack. On the other hand, the reasons why certification revocation is needed can also not be ignored. Therefore it would be wise to allow for “minimal" usage of certificate revocation.

[12] $\boldsymbol{c_{0..j}}$ is a certificate chain where $\boldsymbol{c_0}$ is the certificate from the root certificate authority, $\boldsymbol{c_j}$ is the certificate of the client machine and all the intermediate certificates are of the other CA's

What does one do when his certificate expires?

1. Get a new certificate issued from the CA, for the ssh server.
2. Alternatively, get the old certificate re-certified by sending it to the CA.

“Issuing a certificate” means that the issuer is attesting that a given party is the exclusive entity that knows the private key matching *that* public key, hence tasks like renewing certificates are not trivial in nature as they play a fundamental role in developing a robust infrastructure.

We opted for the second option. The reason for that is that it offers consistency and minimizes risks due to replay attacks.

How does the client machine get the server certificate verified?

In a complex Public Key Infrastructure, the client needs to know the "certification path" of certificates taking the server's public key to the "root" public key that the client knows. The suitability of the following technique in the context of the SSH2 protocol was evaluated and then rejected.

1. Delegated path discovery and validation [11,12],

The client must be able to validate the signer and all the entities vouching for the certificate's validity. The DPV/DPD protocols provide a new approach to certificate validation. The DPV protocol allows clients to dump their validation duties onto a DPV server, which validates a certificate by doing all the necessary homework - including checking CRLs and CRL change reports - using the Online Certificate Status Protocol (OCSP). OCSP is a request/response mechanism for checking the status of a specific certificate.

Using the DPD protocol, clients can query a DPD server to collect and pass on all the information the client needs for local validation. In other words, the client sends certificates to the server, which returns current information about the certificates and any related certificates, such as if it is valid or not [12].

2. Our Approach.

Our approach is simple and extends the SSH2 protocol easily. It has no dependency on an external server other than the ldap server. It also limits the interaction with a certificate authority.

5. IMPLEMENTATION

The fully decentralized solution described in the paper was completely implemented. We also tested the following components of the semi-centralized solution to ensure that the proposed semi-centralized solution was workable:

- Installed LDAP server,
- Hooked up the server with a dB database,
- Populated a table in the database,
- Retrieved a field of the table from the dB through the LDAP server,

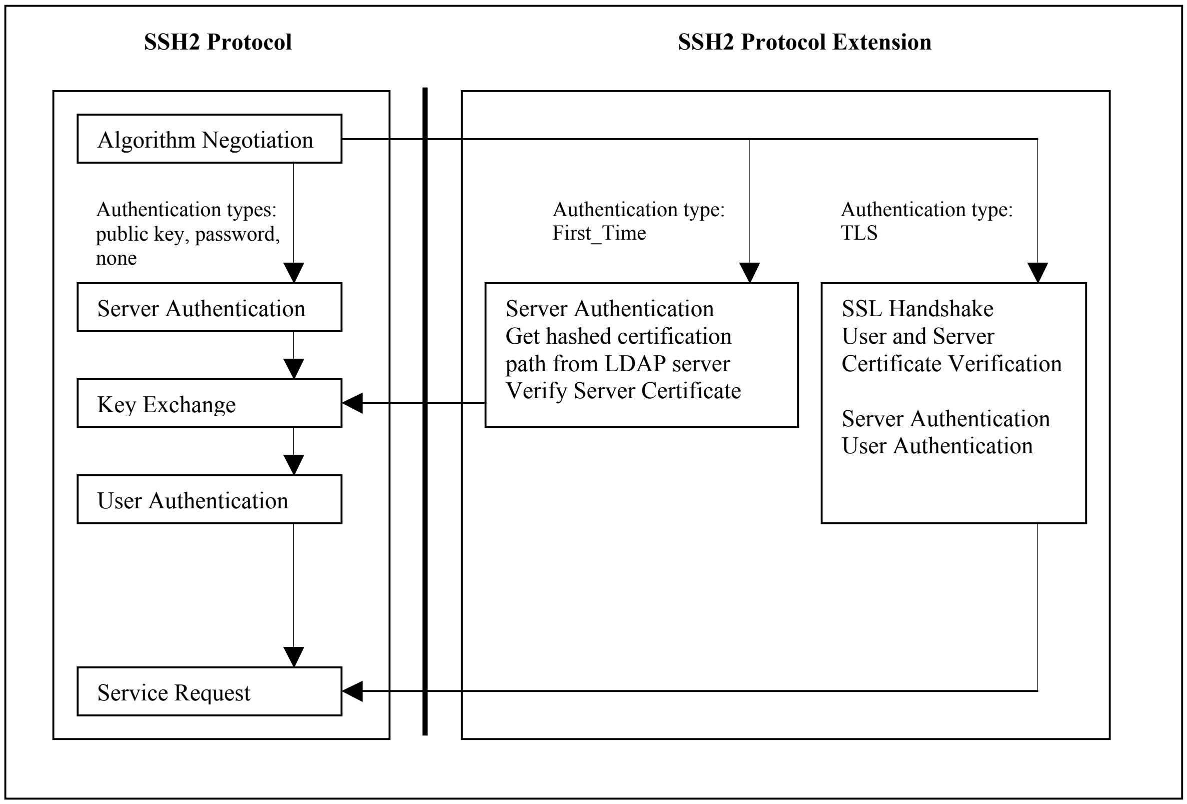
The TLS extension which essentially replaced the SSH handshake with a TLS handshake using certificates was *not* implemented.

Extending SSH to include this authentication method:

The existing SSH2 protocol supported by OpenSSH has the bare skeletal structure to support certificates that does not support certificate verification. In addition, ssh servers are already using OpenSSL libraries for encryption, compression and public key authentication to perform "ssh handshake".

Following is an Architectural Block diagram that explains how we extend ssh protocol to support new authentication mechanisms. Authentication Types TLS and First_Time have been added into the architecture.

Figure 5 Extended SSH2


SSH2 Protocol
SSH2 Protocol Extension
Algorithm Negotiation
Authentication types: public key, password, none
Server Authentication
Key Exchange
User Authentication
Service Request
Authentication type: First_Time
Authentication type: TLS
Server Authentication Get hashed certification path from LDAP server Verify Server Certificate
SSL Handshake User and Server Certificate Verification
Server Authentication User Authentication

5A.Application Interface

There are two open source ssh clients that are being used commonly.

1. OpenSSH Client and Server (for the Linux World).
2. TeraTerm SSH Client (for the Windows platforms)

The only fully deployed open source SSH server is provided by OpenSSH and is distributed freely with Red Hat Linux Operating System. NetworkSimplicity.com provides windows' port for OpenSSH client and server. The drawback of the OpenSSH source code is that it does not provide a graphical user interface.

TeraTerm SSH clients that are interoperable with OpenSSH servers, provide a Graphical User Interface that makes it easy to use on windows platform.

TeraTerm SSH client is an extension to TeraTerm Telnet client, which is also open source.

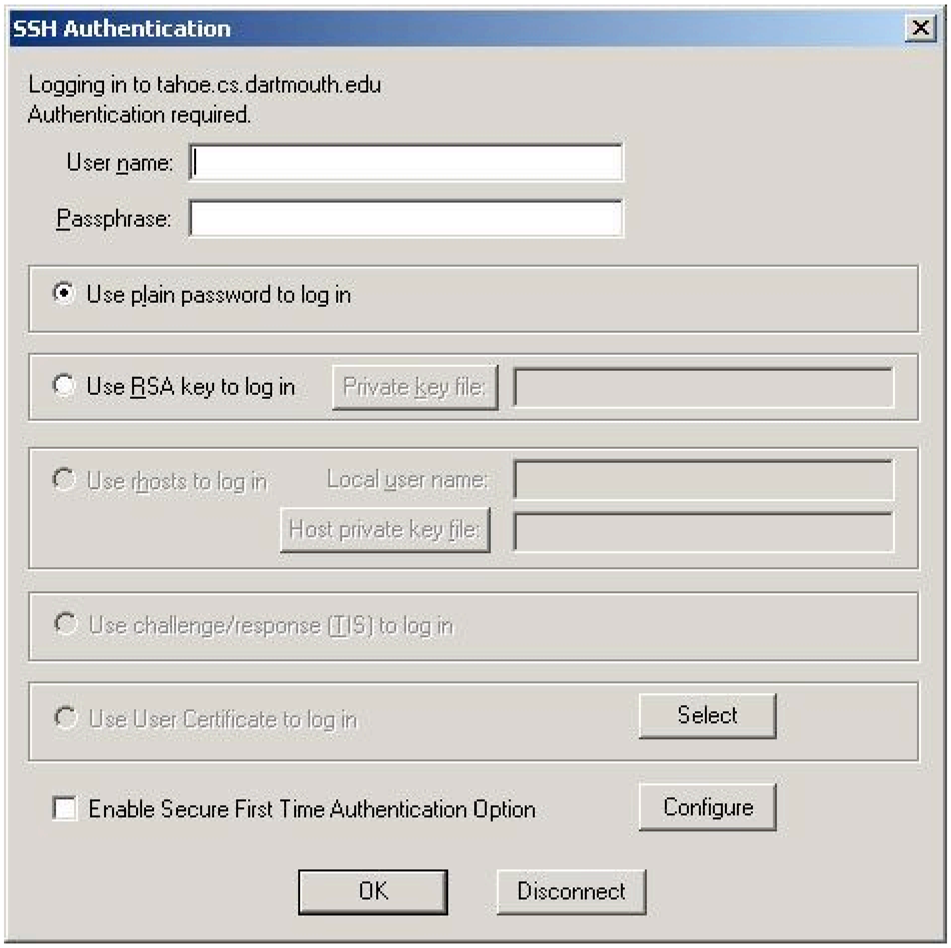
Our extension was added to the TeraTerm SSH client, which is a Windows client, primarily due to the following reason:

- Most of the users at Dartmouth either use Windows or Macintosh Operating Systems.

Following are some of the Graphical User Interfaces that are displayed once the user selects to use ssh. A brief description of the functionality is also provided.

In Figure 6, the user selects one of the radio buttons. If he is a traveling user and is doing authentication for the first time on the client machine, he selects "Enable first time authentication". Then he clicks on the Configure button to download the hashstore from a website or an ldap server.

Figure 6 Authentication Setup



SSH Authentication

Logging in to tahoe.cs.dartmouth.edu
Authentication required.

User name:

Passphrase:

(●) Use plain password to log in

( ) Use RSA key to log in — Private key file:

( ) Use rhosts to log in — Local user name: — Host private key file:

( ) Use challenge/response (TIS) to log in

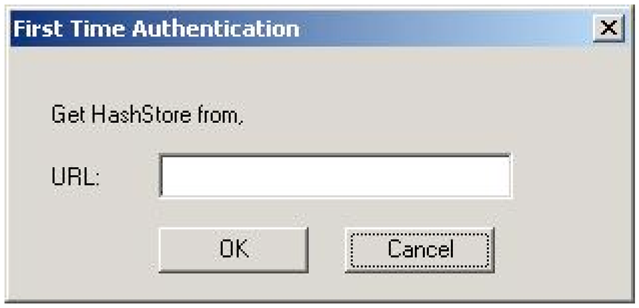
( ) Use User Certificate to log in — Select

☐ Enable Secure First Time Authentication Option — Configure

OK — Disconnect

In Figure 7, the user enters the path to the http server, with the filename for the hashstore. An HTTP connection is made to the http server, and the file is retrieved if it is found. Incase an ldap server name is provided then the connection is made to the ldap server and it is queried for the needed data. As mentioned earlier the "ldap" component was independently tested but was not embedded into this solution

Figure 7 Retrieve Hash Store

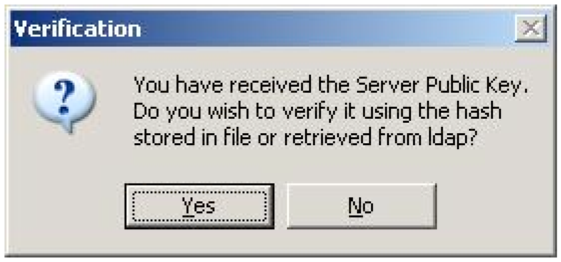


Get HashStore from,

URL:

Once the user has successfully retrieved the hash store, he initiates the ssh connection. He provides the server name and waits on the Authentication setup window while he retrieves the public key of the server for the first time. When the key is retrieved, the user is prompted with the following figure.

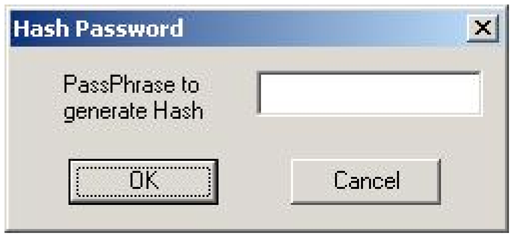
Figure 8 Verify server public key received



You have received the Server Public Key. Do you wish to verify it using the hash stored in file or retrieved from ldap?

If the user clicks ‘No’ then the SSH protocol proceeds as if nothing was modified and the user would be vulnerable to the “man in the middle attack”. However, if the user clicks on ‘Yes’, then he would be prompted to enter the pass phrase to generate the HMAC hash.

Figure 9 Enter Pass Phrase

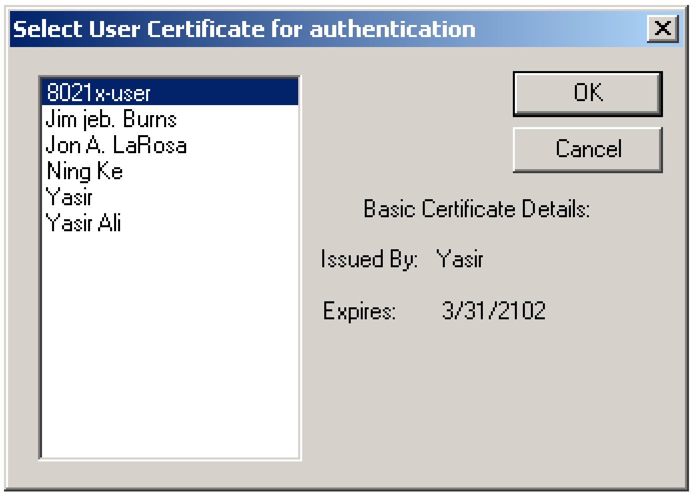


PassPhrase to generate Hash

If the two hashes match (hash generated and the hash stored in the hashstore), then the public key in the keyblob received from the ssh server is verified and the SSH2 protocol proceeds to establish a shared session key. The rest of the SSH2 protocol is the same.

If the user selects the "Use User Certificate to log in" option on Authentication Setup window, then he has to select a user certificate that is stored in the "Personal" Microsoft Certificate Store. In the current implementation, the user public key blob is filled up by taking the public key from the user certificate in case of public key authentication. This feature has been implemenedt but not fully tested yet.

Figure 10 Certificate Selection Window


Select User Certificate for authentication

8021x-user
Jim jeb. Burns
Jon A. LaRosa
Ning Ke
Yasir
Yasir Ali

Basic Certificate Details:

Issued By: Yasir

Expires: 3/31/2102

6. SCOPE OF FUTURE WORK

Following modifications to the graphical interface and functionality were suggested to enhance usability of the prototype:

1. User should be prompted right in the beginning, if he wants to use the conventional SSH protocol or the modified version,
2. If the user selects the modified version then no warning messages should pop up after server key verification,

3. The hashstore should be retrieved for every ssh session from the website entered by the user. The hashstore would not live on the borrowed machine and would be deleted once the ssh session disconnects.

The solution presented in this paper primarily deal with the "Man in the Middle Attack". However, as discussed in the paper, in the existing SSH2 protocol, once the server is verified, the user sends his username and password as *plaintext* in the encrypted channel. Although the protocol is no longer vulnerable to the man in the middle attack, it allows for the possibility of a replay attack if the server's public key changes. To further *strengthen the SSH2 protocol*, user authentication should be replaced with a Challenge Response Authentication method, such as MSCHAPv2. The TTLS protocol specifically allows for that, so should the SSH2. Also SSH Client should be integrated to Certificate Stores provided by browsers such as IE, so that the users can easily install CA certificates on their machines that verify server certificates. The foundations for that work has already been laid in our thesis work.

In the future the wireless networks would merge into the wired networks, offering an acceptable level of security for both. Currently the wireless gurus are looking at the SSH2, IPSec, TLS, TTLS, PEAP and 8021x protocols for authentication. If the existing vulnerabilities of the SSH2 are resolved and an approach is developed that requires *no maintenance* by the user without necessitating the deployment of a Certificate Authority, it would make it a top candidate for authentication and *tunneling* in both the wired and wireless networks. Therefore, the feasibility of the SSH2 for wireless authentication should also be evaluated so that it can be modified to better fit a wireless network as well.

All the code that was used and modified in this project is open source and can be downloaded from http://www.cs.dartmouth.edu/~yasir/Thesis/. The binaries for the modified TeraTerm ssh clients are also available. The code contains all the needed OpenSSL libraries, crypt32 libraries from Microsoft platform SDK needed for certificate management. A readme.txt file explains how to setup the client, and the hash generation tool. There needs to be some more work done on the ssh server with regards to certificate verification using CA Certification Path. Individuals who may be interested in continuing with this work can contact the PKI Lab at ISTS, Dartmouth.

7. RELATED WORK AND CONCLUSION

Developing a Public Key Infrastructure to be used with OpenSSH in an academic environment expose us to particularly interesting and different scenarios where we have to take into account the design considerations as discussed in the previous sections of this paper. Following commercial SSH products also offer PKI based solutions.

7A. F-SSH:

An **F-Secure** product, ***F-SSH*** claims to have the requisite design to be usable with Certificate Authorities. However they have not developed a toolkit or a product that incorporates PKI as yet. The authentication method that they support which can provide Certificate verification is PAM (pluggable Authentication Mode). However, PAM is typically used with smart cards.

7B. SSH Client 3.2.0, SSH Communications Security Inc:

The **SSH Communications Security Inc.** have developed a suite of Applications such as:

- ***SSH Client***
- ***SSH Certifiers***
- ***SSH Certification Toolkits***.

SSH provides their own suit of products that allows for certificate based mutual authentication.

SSH Client by SSH Inc. is aimed to ‘do it all’ that may be needed for a Public Key Infrastructure. It discovers CA on a LAN, checks CRL’s and handles all the related functionality of generating a user certificate and installing it and also installing the certification paths. One of the major drawbacks of their solution is that a user cannot use password authentication and also perform the new server authentication mechanism other than the local database verification that they support.

Keeping in view the following factors:

- Ease of management and deployment,
- Scalability: It should work equally well with server machines within the same network or server machines across different domains,
- Modularized addition to the existing open source OpenSSH. We add an independent module into the SSH client
- Open source so that it is available to everyone and anyone, within the export restrictions,
- And most importantly, using usernames and passwords, instead of user certificates,

Our solution is unique in its nature as it is simple and embeds OpenSSH into PKI in a manageable way, even for an academic setting as big as the one at Dartmouth. Our solution strengthens SSH against "the Man in the middle attack". We present two approaches, a centralized approach that requires a CA and a decentralized one that does not require a CA. The ease of use and deployment of the "decentralized non CA approach" distinguishes our solution from the one provided by SSH Inc, which is based on a PKI Infrastructure with Certificate Authorities. Given the current state of PKI, as both the academia and the industry strive to develop internet drafts for standardized protocols that would make the deployment and management of such infrastructure relatively manageable, the fact of the matter remains that currently there does not exist a homogenous, easy to deploy Public Key Infrastructure mechanism. Most of the small corporate settings do not need to invest and develop a hierarchical trust model for a PKI. Their needs are simple. The users want to connect to a "few" machines remotely. They create VPN to access their corporate network or use SSH to access data securely over connections. Our decentralized approach is ideal for such small-scale corporate environments. The users do not have to learn the how to use user certificates and the system administrators do not have to setup a PKI. Our centralized CA approach is suited for larger networks with several users. It develops a PKI infrastructure by having the minimal set of components that are needed to set up. Design constraints and policies are set to achieve scalability with minimal set up effort that would require minimal management effort as well. Unlike SSH Inc, our PKI design does not require the user to interface with the CA, which makes it more secure. To sum up, our solution makes ssh usage more secure for both, small and large networks.

8. REFERENCES

[1] Carlisle Adams and Robert Zuccherato, “A General, Flexible Approach to Certificate Revocation,” Entrust technologies White Paper, June 1998.

[2] Carlisle Adams and Stephen Farrell, “Internet X.509 Public Key Infrastructure Certificate Management Protocols,” IETF RFC 2510, March 1999.

[3] Daniel J. Barrett & Richard E. Silverman, SSH The Secure Shell, The Definitive Guide, O'REILLY & Associates, 2001

[4] S. Boeyen, T. Howes, P. Richard, “Internet X.509 Public Key Infrastructure, LDAPv2 Schema”, RFC 2587, June 99

[5] Markus Friedl, et al., “Diffie-Hellman Group Exchange for the SSH Transport Layer Protocol”, Internet Draft, January 2002.

[6] H. Krawczyk, M. Bellare, R. Canetti, “HMAC: Keyed Hashing for Message Authentication, RFC 2104, February 1997

[7] Ninghui Li and Joan Feigenbaum, "Nonmonotonicity, User Interfaces, and Risk Assessment in Certificate Revocation ", in Proceedings of the Fifth International Conference on Financial Cryptography 2001, Springer-Verlag NCS.

[8] Steve Lloyd, David Fillingham, Richard Lampard, Steve Orlowski, John Weigelt, “CA-CA Interoperability”, White Paper on PKI Forum, March 2001.

[9] Michael Myers, “Revocation: Options and Challenges,” in Financial Cryptography 1998, Springer-Verlag NCS.

[10] Michael Myers, Rich Ankney, Carlisle Adams, Stephen Farrell, and Carlin Covey, “Online Certificate Status Protocol, version 2”, Internet Draft, March 2001.

[11] Denis Pinkas, Russ Housley, “Delegated Path Validation and Delegated Path Discovery Protocol Requirements”, Internet Draft, February, 2002.

[12] Denis Pinkas, Russ Housely, “Certificate Validation Protocol”, Internet Draft, June, 2002

[13] Dawn Xiaodong Song, David Wagner, Xuqing Tian, “Timing Analysis of Keystrokes and Timing Attacks on SSH”, In 10th USENIX Security Symposium, 2001.

[14]

i. T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen, “SSH Protocol Architecture”, Network Working group, Internet Draft, January 31, 2002.

ii. “SSH Connection Protocol”, Network Working group, Internet Draft, January 31, 2002. “SSH Transport Layer Protocol”, Network Working group, Internet Draft, January 31, 2002.

iii. "Authentication Protocol", Network Working group, Internet Draft, February 28, 2002.

[15] Yasir Ali and Sean W. Smith, "Flexible and Scalable Public Key Security for SSH ", submitted for the Second Annual PKI Workshop, Feb 2003

[16] Referenced Websites:

i. www.SSH.com
ii. www.OpenSSH.org
iii. www.securityportal.com
iv. http://www.openldap.org/
v. http://www.networksimplicity.com/openssh/
vi. http://www.f-secure.com/
vii. http://www.webopedia.com/TERM/P/PKI.html

[17] Email correspondence with Jeff Schiller and Robert O'Callahan.

9. Appendix - Code Snippets Used to Extend TTSSH.

- Getting hashstore from the web interface.

```
#include "stdafx.h"
#include "afxinet.h"


#ifdef __cplusplus
extern "C" {
#endif

BOOL GetFileFromWeb ( char* strServerName2,
                     char* URL
                   )
{
      CString           strObject;
      INTERNET_PORT     nPort;
      DWORD             dwServiceType;
      DWORD    dwAccessType = INTERNET_OPEN_TYPE_PRECONFIG;
      CInternetSession  session("Tera Term", dwAccessType);
      CHttpConnection*  pServer = NULL;
      CHttpFile*        pFile = NULL;
      CString strServerName(strServerName2);

      AfxParseURL( URL,
                     dwServiceType,
                         strServerName,
                         strObject,
                         nPort);

      pServer = session.GetHttpConnection(strServerName, nPort);
      pFile = pServer->OpenRequest(
                         CHttpConnection::HTTP_VERB_GET,
                               strObject,
                               NULL,
                               1,
                               NULL,
                               NULL,
                               INTERNET_FLAG_RELOAD);
      pFile->SendRequest();
      DWORD dwStatusCode;
      pFile->QueryInfoStatusCode(dwStatusCode);

      if(dwStatusCode == HTTP_STATUS_OK)
      {
            int len = pFile->GetLength();
            char buf[2000];
            int numread;
            CString filepath =".\\hashstore.txt";

            CFile myfile(filepath,
                  CFile::modeCreate|
                  CFile::modeNoTruncate |
                  CFile::modeWrite|
                  CFile::typeBinary);
      while ((numread = pFile->Read(buf,sizeof(buf)-1)) > 0)
            {
                  buf[numread] = '\0';
                  myfile.Write(buf, numread);
             }
            myfile.Close();
            MessageBox(NULL,"Hash file retrieved successfully!",
                      "Error",MB_OK);
      }
      else
            MessageBox(NULL,"File Not retrieved!","Error",MB_OK);
      pFile->Close();
      delete pFile;
      pServer->Close();
      delete pServer;
      session.Close();

      return TRUE;
}

#ifdef __cplusplus
}
#endif
```

- Certificate Manipulation functions.

```
#include "stdafx.h"
#include "certificates.h"
#include <wincrypt.h>
#ifdef __cplusplus
extern "C" {
#endif

static int getCN(PCERT_NAME_BLOB name, LPTSTR str, DWORD len)
{
      PCERT_RDN_ATTR pAttr;
      BYTE *buffer = NULL;
      ULONG sz = 0;
      int res = 1;

      if (!CryptDecodeObject(
            X509_ASN_ENCODING,
            X509_NAME,
            name->pbData,
            name->cbData,
            CRYPT_DECODE_NOCOPY_FLAG,
            NULL,
            &sz))
      {
            res = 0;
            goto end;
      }

      if ((buffer = ( BYTE *)GlobalAllocPtr( GMEM_MOVEABLE |
                                  GMEM_ZEROINIT, sz )) == NULL)
      {     res = 0;
```

```
			goto end;
		}

		if (!CryptDecodeObject(
			X509_ASN_ENCODING,
			X509_NAME,
			name->pbData,
			name->cbData,
			CRYPT_DECODE_NOCOPY_FLAG,
			buffer,
			&sz))
		{
			res = 0;
			goto end;
		}

		if ((pAttr = CertFindRDNAttr(
			szOID_COMMON_NAME,
			(PCERT_NAME_INFO)buffer)) != NULL)
		{
			res = CertRDNValueToStr(
				pAttr->dwValueType,
				&pAttr->Value,
				str,
				len);
		} else
			res = 0;

end:
		if (buffer != NULL)
			GlobalFreePtr(buffer);

		return res;

}

void freeCertDescription(CertDescription *cd)
{

		if (cd == NULL)
			return;
		if (cd->expDate != NULL)
			GlobalFreePtr(cd->expDate);
		if (cd->friendlyName != NULL)
			GlobalFreePtr(cd->friendlyName);
		if (cd->hash.pbData != NULL)
			GlobalFreePtr(cd->hash.pbData);
		if (cd->issuedBy != NULL)
			GlobalFreePtr(cd->issuedBy);
		if (cd->issuedTo != NULL)
			GlobalFreePtr(cd->issuedTo);
		GlobalFreePtr(cd);
  }


CertDescription *getCertDescription(PCCERT_CONTEXT pCertContext)
{
      ULONG sz;
      SYSTEMTIME sysTime;
      CertDescription *cd = (CertDescription *)GlobalAllocPtr(
        GMEM_MOVEABLE | GMEM_ZEROINIT, sizeof(CertDescription) );

      if ((sz =(ULONG)getCN(&pCertContext->pCertInfo->Subject,
           NULL, 0)) > 0)
      {
            sz *= sizeof(TCHAR);
            cd->issuedTo = ( LPTSTR )GlobalAllocPtr(
                            GMEM_MOVEABLE | GMEM_ZEROINIT, sz);
            getCN(&pCertContext->pCertInfo->Subject,cd->issuedTo,
                  sz);
      }

      if ((sz = (ULONG) getCN(&pCertContext->pCertInfo->Issuer,
            NULL, 0)) > 0)
      {
            sz *= sizeof(TCHAR);
            if ((cd->issuedBy = ( LPTSTR )GlobalAllocPtr(
                 GMEM_MOVEABLE | GMEM_ZEROINIT, sz)) == NULL)
                  goto err;
            getCN(&pCertContext->pCertInfo->Issuer, cd->issuedBy,
            sz);
      }

      FileTimeToSystemTime(&pCertContext->pCertInfo->NotAfter,
                          &sysTime);
      if ((sz = GetDateFormat(
            LOCALE_USER_DEFAULT,
            DATE_SHORTDATE,
            &sysTime,
            NULL,
            NULL,
            0)) > 0)
      {


if ((cd->expDate = ( LPTSTR )GlobalAllocPtr( GMEM_MOVEABLE|
                    GMEM_ZEROINIT, sz * sizeof(TCHAR) )) == NULL)
                    goto err;
      GetDateFormat(
                  LOCALE_USER_DEFAULT,
                  DATE_SHORTDATE,
                  &sysTime,
                  NULL,
                  cd->expDate,
                  sz);
      }

      sz = 0;
      CertGetCertificateContextProperty(
            pCertContext,
            CERT_FRIENDLY_NAME_PROP_ID,
            NULL, &sz);
      if (sz > 0)
     {
      if ((cd->friendlyName = ( LPWSTR )GlobalAllocPtr(
          GMEM_MOVEABLE | GMEM_ZEROINIT, sz )) == NULL)
                  goto err;
      if (!CertGetCertificateContextProperty(
                  pCertContext,
                  CERT_FRIENDLY_NAME_PROP_ID,
                  cd->friendlyName, &sz))
                  goto err;
      }

      cd->hash.cbData = 0;
      CertGetCertificateContextProperty(
            pCertContext,
            CERT_HASH_PROP_ID,
            NULL, &cd->hash.cbData);
      if (cd->hash.cbData > 0) {
            if ((cd->hash.pbData = ( PBYTE )GlobalAllocPtr(
                 GMEM_MOVEABLE | GMEM_ZEROINIT,
                 cd->hash.cbData )) == NULL)
                  goto err;
            if (!CertGetCertificateContextProperty(
                  pCertContext,
                  CERT_HASH_PROP_ID,
                  cd->hash.pbData, &cd->hash.cbData))
                  goto err;
      }

      return cd;

err:
      if (cd != NULL)
            freeCertDescription(cd);

      return NULL;

}
#ifdef __cplusplus
}
#endif
```

- **Hashing Code at the Client End**

```
#include <stdio.h>
#include <windows.h>
#include <wincrypt.h>
#include "ttxssh.h"
#include <openssl/ssl.h>
#include <openssl/hmac.h>

#define   MY_ENCODING_TYPE      (PKCS_7_ASN_ENCODING   |
X509_ASN_ENCODING)
#define ArrayLen(a)  (sizeof (a) / sizeof (a[0])) +1
FILE *stream;
FILE *stream2;
extern CHAR HPassPhrase[128];
extern void HandleError(char *s);
extern BOOL GetandCompareHash( unsigned char* szIPandHash );
extern BOOL puthHash( unsigned char* szIPandHash );
extern void verifyhashing(PTInstVar pvar,
                                       int bits,
                                       unsigned char FAR * exp,
                                       unsigned char FAR * mod )
{

      HMAC_CTX ctx;
      CHAR MsgDigest[64];
      UCHAR *temp = exp;
      UCHAR *temp2 = exp;
      int i = 0;
      int count = 0;
      unsigned int  MsgDigestLen = 0;
      UCHAR szHost[128] = "";
      UCHAR cleanexp[4096] = "";
      DWORD lpdwResult = 0;

      SendMessage(pvar->NotificationWindow,
                    WM_COMMAND,
                        ID_HASHPASSPHRASE,
                        0);

      for (i =0; i<4;i++)
      {
            while ( (UCHAR)*temp != ' ')
            {
                  *temp++;
                  count ++;
            }
            *temp++;
            count ++;
      }
      strncpy(cleanexp,exp,count-1);

      HMAC_Init(&ctx,HPassPhrase,strlen(HPassPhrase), EVP_md5());
      HMAC_Update(&ctx, cleanexp, strlen(cleanexp) );
      HMAC_Final(&ctx, MsgDigest, &MsgDigestLen);

      count = 0;
      for (i =0; i<3;i++)
      {
            while ( (UCHAR)*temp2 != ' ')
            {
                  *temp2++;
                  count ++;
            }
            *temp2++;
            count ++;
      }

      strncpy(szHost,exp,count-1);

      if ( szHost == NULL )
            exit(1);

      strcat(szHost," MD:");
      strncat(szHost,MsgDigest,strlen(MsgDigest) -1 );
      if ( GetandCompareHash(szHost))
      {
            MessageBox(NULL,
            "Hashes Matched. Server public key verified.",
            "Verification Successful",
            MB_ICONINFORMATION | MB_OK);
      }

      else
      {
            MessageBox(NULL,
            "Server public key verification failed.\n"\
            "Possible Reasons:\n"\
            "1. Public Key Hash did not match.\n"\
            "2. Hashing password used is incorrect.\n"\
            "3. HashStore does not contain the entry for host
               name provided",
            "Verification Unsuccessful",
            MB_ICONINFORMATION | MB_OK);
            exit(1);
      }

} // End of HASHING

void HandleError(char *s)
{
    MessageBox(NULL,s,"Error",MB_OK);
      GetLastError();
    exit(1);
}

BOOL GetandCompareHash( unsigned char* szIPandHash )
{
      char buffer[4096];
      if( (stream  = fopen( "hashstore.txt", "r+" )) == NULL )
            exit(1);

      fread( buffer, sizeof( unsigned char ), 4096, stream );

      if ( strstr( buffer, szIPandHash ) != NULL )
      {
            fclose( stream );
            return TRUE;

      }
      return FALSE;
}
```

- **Code for Handling the public key when received.**

```
static BOOL handle_server_public_key(PTInstVar pvar) {
  int server_key_public_exponent_len;
  int server_key_public_modulus_pos;
  int server_key_public_modulus_len;
  int host_key_bits_pos;
  int host_key_public_exponent_len;
  int host_key_public_modulus_pos;
  int host_key_public_modulus_len;
  int protocol_flags_pos;
  int supported_ciphers;
  char FAR * inmsg ;
  char FAR * dest = (char FAR * ) malloc( 2048 );
  int len = 0;

     if (!grab_payload(pvar, 14)) return FALSE;
  server_key_public_exponent_len = get_mpint_len(pvar, 12);

  if (!grab_payload(pvar, server_key_public_exponent_len + 2))
return FALSE;
  server_key_public_modulus_pos = 14 +
server_key_public_exponent_len;
  server_key_public_modulus_len = get_mpint_len(pvar,
server_key_public_modulus_pos);

  if (!grab_payload(pvar, server_key_public_modulus_len + 6))
return FALSE;
  host_key_bits_pos = server_key_public_modulus_pos + 2 +
server_key_public_modulus_len;
  host_key_public_exponent_len = get_mpint_len(pvar,
host_key_bits_pos + 4);

  if (!grab_payload(pvar, host_key_public_exponent_len + 2))
return FALSE;
  host_key_public_modulus_pos = host_key_bits_pos + 6 +
host_key_public_exponent_len;
  host_key_public_modulus_len = get_mpint_len(pvar,
host_key_public_modulus_pos);

  if (!grab_payload(pvar, host_key_public_modulus_len + 12))
return FALSE;
  protocol_flags_pos = host_key_public_modulus_pos + 2 +
host_key_public_modulus_len;

  inmsg = pvar->ssh_state.payload;
  CRYPT_set_server_cookie(pvar, inmsg);

  if (!CRYPT_set_server_RSA_key(pvar, get_uint32(inmsg + 8),
pvar->ssh_state.payload + 12,
    inmsg + server_key_public_modulus_pos)) return FALSE;

  if (!CRYPT_set_host_RSA_key(pvar, get_uint32(inmsg +
host_key_bits_pos),
    inmsg + host_key_bits_pos + 4,
    inmsg + host_key_public_modulus_pos)) return FALSE;
  pvar->ssh_state.server_protocol_flags = get_uint32(inmsg +
protocol_flags_pos);

  supported_ciphers = get_uint32(inmsg + protocol_flags_pos +
4);
  if (!CRYPT_set_supported_ciphers(pvar, supported_ciphers,
supported_ciphers)) return FALSE;
  if (!AUTH_set_supported_auth_types(pvar, get_uint32(inmsg +
protocol_flags_pos + 8))) return FALSE;
```

```
   /* this must be the LAST THING in this function, since it
can cause host_is_OK to be called. Used specifically for first
time authentication*/
  HOSTS_check_host_key(pvar, pvar->ssh_state.hostname,
    get_uint32(inmsg + host_key_bits_pos),
    inmsg + host_key_bits_pos + 4,
    inmsg + host_key_public_modulus_pos);

  return FALSE;
}



BOOL HOSTS_check_host_key(PTInstVar pvar, char FAR * hostname,
                          int bits, unsigned char FAR * exp,
                         unsigned char FAR * mod)
{
  int found_different_key = 0;

  if (pvar->hosts_state.prefetched_hostname != NULL
    && stricmp(pvar->hosts_state.prefetched_hostname,
       hostname) == 0
    && match_key(pvar, bits, exp, mod)) {
    SSH_notify_host_OK(pvar);
    return TRUE;
  }

  if (begin_read_host_files(pvar, 0))
  {
    do {
      if (!read_host_key(pvar, hostname, 0)) {
        break;
      }

      if (pvar->hosts_state.key_bits > 0) {
        if (match_key(pvar, bits, exp, mod)) {
          finish_read_host_files(pvar, 0);
          SSH_notify_host_OK(pvar);
          return TRUE;
        } else {
          found_different_key = 1;
        }
      }
    } while (pvar->hosts_state.key_bits > 0);

    finish_read_host_files(pvar, 0);
  }

  pvar->hosts_state.key_bits = bits;
  pvar->hosts_state.key_exp = copy_mp_int(exp);
  pvar->hosts_state.key_mod = copy_mp_int(mod);
  free(pvar->hosts_state.prefetched_hostname);
  pvar->hosts_state.prefetched_hostname = _strdup(hostname);

    if(MessageBox(NULL,
                "You have received the Server Public Key. \n"
                "Do you wish to verify it using the hash \n" \
                "stored in file or retrieved from ldap?",
                "Verification",
                MB_ICONQUESTION | MB_YESNO)
                == IDYES)
  {
     Authflag = 1;
     //CRYPT_get_server_key_info(pvar, dest, len);
     if ( pvar->crypt_state.server_key.RSA_key != NULL)
     {
            char FAR * keydata = format_host_key(pvar);
                 verifyhashing(pvar,
                 pvar->hosts_state.key_bits,
                 keydata,
                 pvar->hosts_state.key_mod );
     }
     else
           MessageBox(NULL,"No RSA key","Error",MB_OK);
  }

  if (found_different_key) {
    PostMessage(pvar->NotificationWindow,WM_COMMAND,
                ID_SSHDIFFERENTHOST, 0);
    }
  else {
    PostMessage(pvar->NotificationWindow,
                WM_COMMAND, ID_SSHUNKNOWNHOST, 0);
  }
  return TRUE;
}
```

- **Linux Configurator Code.**

```
/* gcc -o configurator -I../include configurator.c
    ../libcrypto.a

For most of the machines:
gcc  -o  configurator  -I/usr/local/ssl/include  configurator.c
usr/local/ssl/libcrypto.a
You have to locate the ssl include and lib folders to compile
on linux.

 */
#include <stdio.h>
#include <openssl/ssl.h>
#include <openssl/hmac.h>

FILE *stream;
FILE *stream2;

int puthHash( unsigned char* exp )
{
   HMAC_CTX ctx;
   char szPassword[128] = "";
   char MsgDigest[64];
   unsigned char *temp = exp;
   unsigned char *temp2 = exp;
   int i = 0;
   int count = 0;
   unsigned int  MsgDigestLen = 0;
   unsigned char szHost[128] = "";
   unsigned char cleanexp[4096] = "";

   for (i =0; i<4;i++)
   {
         while ( (unsigned char)*temp != ' ')
         {
               *temp++;
               count ++;
         }
         *temp++;
         count ++;
   }
   strncpy(cleanexp,exp,count-1);

   printf("\nEnter the passphrase to hash:\n");
   scanf("%s", &szPassword );

   HMAC_Init(&ctx, szPassword,strlen(szPassword), EVP_md5());
   HMAC_Update(&ctx, cleanexp, strlen(cleanexp) );
   HMAC_Final(&ctx, MsgDigest, &MsgDigestLen);


   count = 0;
   for (i =0; i<3;i++)
   {
         while ( (unsigned char)*temp2 != ' ')
         {
               *temp2++;
               count ++;
         }
         *temp2++;
         count ++;
   }
   strncpy(szHost,exp,count-1);
   if( (stream2  = fopen( "hashstore.txt", "a" )) == NULL )
         return -1;
   if ( szHost == NULL )
         return -1;

   printf("\n---Hash stored in hashstore.txt---\n");
    fprintf(stream2,"%s ",szHost);
   printf("%s ",szHost);
   fprintf(stream2,"MD:%s \r\n",MsgDigest);
   printf("MD:%s \n",MsgDigest);
    fclose(stream2);
   return 0;
}

int main ()
{
   unsigned char buffer[4096];
   unsigned char szHost[4096];

   printf("This program generates the HMAC Hashes\n"
          "for the public keys of host machines.\n"
         "This program is to be run on the local machine\n"
         "whose public key is to be hashed\n\n");
   if( (stream  = fopen( "/etc/ssh/ssh_host_key.pub",
                         "r" )) == NULL )
         return -1;

   fread( buffer, sizeof( unsigned char ), 4096, stream );
   fclose(stream);

   printf("Enter the hostname of the machine:\n");

   scanf("%s", &szHost );
   strcat(szHost," ");
   strcat(szHost,buffer);
   printf("----String to be Hashed---\n%s", szHost);

   puthHash( szHost );

   return 0;
}
```