Flexible and Scalable Public Key Security for SSH[⋆]

Yasir Ali and Sean Smith

Department of Computer Science/PKI Lab
Dartmouth College, Hanover NH 03755 USA
yasir.ali@alum.dartmouth.org
sws@cs.dartmouth.edu

Abstract. A standard tool for secure remote access, the SSH protocol uses public-key cryptography to establish an encrypted and integrity-protected channel with a remote server. However, widely-deployed implementations of the protocol are vulnerable to man-in-the-middle attacks, where an adversary substitutes her public key for the server's. This danger particularly threatens a traveling user Bob borrowing a client machine.
Imposing a traditional X.509 PKI on all SSH servers and clients is neither flexible nor scalable nor (in the foreseeable future) practical. Requiring extensive work or an SSL server at Bob's site is also not practical for many users.
This paper presents our experiences designing and implementing an alternative scheme that solves the public-key security problem in SSH without requiring such an a priori universal trust structure or extensive sysadmin work—although it does require a modified SSH client. (The code is available for public download.)

Keywords: SSH, man-in-the-middle.

1 Introduction

In the UNIX world, users traditionally used `telnet` and `ftp` to access remote machines (to establish login sessions and transfer files, respectively). However, these commands transmitted userids and passwords in the clear, and the increasing insecurity of the networks over which these commands operated have made this risk unacceptable.

Consequently, the *secure shell (SSH)* (e.g., [2, 10–13]) has emerged as the de facto replacement for these commands. Rather than typing `telnet` and `ftp` to reach a remote machine S, the user invokes `ssh`, which uses public-key cryptography establish authentication and encrypted communications over unsecured channels. The server presents a public key, and the client machine uses standard cryptography to establish a protected channel with the party knowing the private key—presumably, the server. SSH can even permit the user to authenticate via a key pair instead of a password; however, we conjecture that most users stay with the simpler authentication.

[⋆] This work was supported in part by the Mellon Foundation, by Internet2/AT&T, and by the Office for Domestic Preparedness, U.S. Department of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors. A preliminary version of this paper appeared as Technical Report TR2003-441, Department of Computer Science, Dartmouth College.

However, common SSH implementations overlook an important property: the binding of the server's public key to the identity of the server to which the user intended to connect. This oversight makes the user susceptible to man-in-the-middle attacks, in which the adversary substitutes her public key for the server's; if the user then authenticates via passwords, the adversary can gain complete control of the user's account.

This risk is particularly pronounced in settings where a traveling user is borrowing a client machine that does not *a priori* have a trusted copy of the intended server's public key. We stress that in this model, the user may trust the client machine he or she is borrowing—but not the client machine's network environment. (Indeed, the second author encountered this: a trusted colleague's machine, in an institute suffering DNS attacks.)

Solving these problems in SSH requires introducing a way for SSH clients to securely bind public keys to servers. Solving these problems in the real world requires that any particular SSH client that any particular user wishes to use be able to perform this binding for any particular server the user might want to connect to.

In the long-term, the DNSSEC vision—using PKI to secure all DNS information—would enable a nice solution (e.g., [7]); however, we don't see this happening in the near-term. Perhaps the next natural approach would be to establish a traditional hierarchical PKI for SSH servers; all SSH clients would know the trust root; all SSH servers would have access to a CA/RA system that would sensibly bind the public keys to usable names; trust paths for any given server would somehow arrive at any given client. (Indeed, this is the approach we first considered; and similar commercial offerings have since emerged.)

However, this natural approach does not meet our real world constraints (Sec. 3.2). This universal trust structure needs to be in place before the traveling user can securely connect from a remote machine. Furthermore, many system environments do not provide a natural hierarchy of certifiers or machine names. (For example., one colleague at a corporation "`bar.com`" cited 10^4 machines with names of the form `foo.bar.com`, and whose names were changed apparently at whim by remote sysadmins.)

Alternatively, one might consider a many-rooted trust structure, consisting of many domains linked by bridges and cross-certification. However, it is not reasonable to assume that this solution, attractive in theory, will be workable in wide-scale practice any time soon. (For example, efforts to use bridging to achieve painless interoperability between academic domains academic-to-government domains in the US create ongoing research and engineering challenges.)

Yet another solution might be to build on the "universal PKI" that already exists on desktops: the trust roots built into browsers, and the burgeoning support for browser personal keystores for users. To that end, we considered (and also began prototyping) some additional approaches that used the browser-based SSL PKI to authenticate servers, and possibly clients too. However, this approach would require that all users be affiliated with a site that that sets up and maintains an SSL Web server (and pays for annual renewal of a certificate from a standard browser trust root); while assuming a home Web server was reasonable (and common in many academic and corporate environments), we felt that assuming an SSL server was not. This approaches thus loses flexibility.

This consideration left us with the challenge: how do we bring public-key security to SSH, in a way that provides the flexibility and scalability that can permit easy adoption in the real world, without requiring an a priori trust structure or an SSL server?

Sect. 2 provides the background knowledge to understand the problem. Sect. 3 describes the particular risk and usage scenarios we envisioned, and the design constraints that resulted. Sect. 4 presents the solution we developed (in a sense, a decentralized PKI that requires neither certificates nor CAs). Sect. 5 presents architectural and implementation[1] details. Sect. 6 considers some alternate approaches and future work.

2 Background

2.1 The SSH Protocol

First, we consider the SSH protocol (e.g., [2, 10–13]).

When a user on a client machine tries to establish a secure channel with a remote machine, what really happens is the SSH client (on the client host) carries out this protocol with the *SSH daemon* on the server host.

Put simply, SSH allows these two hosts to construct a secure channel for data communication using Diffie-Hellman key exchange, which 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.

SSH allows for both the server and the client to authenticate using DSA, as part of this exchange. Typically, the client will authenticate the server by comparing a fingerprint of the server's public key with one stored in a file on the client machine; if they do not match, 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. (We conjecture that the typical user would click "OK" and keep going.)

There are three main parts of the SSH protocol: algorithm negotiation, authentication, and 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. Authentication [13] is further broken down in two pieces: key exchange (transport layer) and user authentication (user authentication layer).

The purpose of the key exchange is dual. Firstly, it attempts to authenticate the server to the client. Secondly, it establishes a shared key 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

[1] Prototype source and binaries are available at `http://www.cs.dartmouth.edu/~sws/yasir_thesis`.

server signature received and then continues with user authentication. User authentication methods that are supported and are a part of the SSH protocol include passwords, public key, PAM, and Kerberos.

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.

2.2 Tools

Popular open source tools are the OpenSSH client and server, distributed freely with Red Hat Linux distributions; and the TeraTerm SSH client for Windows. Popular commercial versions include the SSH client and server developed by SSH Inc.

The commercial SSH provides support for a traditional PKI, as we discuss later. However, the OpenSSH client current at the time of our experiments (openssh3.4) did not provide any code that verifies certificates or certificate chains. Rather, certificates were merely treated as public keys. If a key *blob* (the technical name for a data structure that loads the public key) contains a certificate instead of a public key, OpenSSH had 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.

3 Usage Scenarios and Constraints

3.1 Basic Vulnerability

In many public-key applications, the relying party can directly verify that the other entity knows a private key matching a particular public key. However, to draw a more useful conclusion from this, the relying party needs to establish a binding between this public key and some relevant property of this entity.

In the case of SSH, the user needs to establish that the server with whom his client has just established a session is the server to whom he intended to connect.

If the client machine has an *a priori* relationship with the server to which the user wants to connect, under the same server name the user wants to use, and the server's key has not changed, then the user possesses such a binding inductively. (Indeed, many installations suggest that a user traveling with a laptop first `ssh` to his desired home hosts while safely within a trusted home LAN, in order to pre-load the laptop's fingerprint store.)

However, if these things do not hold, then the user is at risk (e.g., see [9]). When the server sends its public key, the user has no way to verify if this key matches the intended server. It is trivial for an attacker to sit in the middle and intercept the connection, and send her own public key and signature instead. The user may then send his own password to the attacker thinking that she is the intended server.

If the client already has the server's public key, the user will generally receive a warning such that "server's key has changed. Continue?" Most users typically hit "Yes" and do not realize the risk. However, if it is the first time the user is connecting to this server from this client, the client will not have the server's public key stored locally, and the user will be none the wiser.

(Researchers have also identified how SSH encryption can protect the plaintext content on a channel but still leak other information [8], but we do not consider those issues here.)

3.2 Real-World Constraints

In the trust model we consider, the user trusts the client machine and his intended remote server to work correctly. However, the user does not trust the server machine he actually connects to until he verifies the server's identity; he also does not trust the client's network environment, including its DNS. If the client does not have a trusted fingerprint of the server already loaded, how does the user establish the binding?

We enumerate some desired goals of an effective solution:

- The solution should enable users from borrowed (but trusted) clients, in untrusted network environments, 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 also accommodate users in domains where no such services exist.
- The solution should *not* require that a new universal PKI structure (neither single-rooted nor multi-rooted) 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.

3.3 Existing Solutions

As mentioned earlier, a natural approach is to assume the existence of a traditional PKI via which any SSH client can authenticate the binding of the server's public key to its identity. Both SSH Communications Security and F-Secure have had commercial offerings in this area, and new commercial offerings continue to emerge.

However, such approaches did not meet our scalability and flexibility constraints. A universal trust structure must exist before the system is useful; before Bob can use an SSH client at Carla's college, Bob's servers must be certified, and Carla's clients must know a certification path from a trust root to that server. Furthermore, not all domains will have the appropriate personnel and organizational structure to establish reliable and usable bindings in the first place.

We could also require that users carry with them some hardware device (such as a smart card or key fob) that remembers the public keys of the servers they wish to interact with. However, we feel this would be overly awkward and expensive, violating the goals; also, sufficiently robust software and hardware support for such tokens has not permeated the current infrastructure, violating the "small delta" constraint.

4 Our Solution

4.1 Overview

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 can neither hard-code one trust root into all clients, nor assume that each client will have a local trust root with a path to Bob's server. The "no memorization" constraint means that the user cannot bring it with them.

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 to 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?

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', k' 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 that M has not been altered in transit. (In a sense, this like a digital signature, except we have a symmetric key instead of a key pair, and thus we lose non-repudiation.)

Our constraints dictate that we cannot force the user to memorize a public key—but users easily memorize URLs and passphrases. Our general solution thus has two parts. First, we built a *modified SSH client* that accepts a location (such as URL) and passphrase from the user; the client then fetches the the trust root from that URL, and verifies its integrity by deriving a symmetric key from the passphrase and using that to check a keyed MAC on the root data. We then built *configurator* tools to create these MAC'd trust roots in the first place.

4.2 Approaches

We now discuss a range of solutions this basic strategy permits.

Decentralized At one extreme, we imagine a user who wants to securely access his standard machines, but does not have a support organization willing or able to take on the job of certifying each SSH daemon.

Our approach permits a fully decentralized SSH PKI that supports such users, by permitting each one to be their own CA.

When the user is at his home site and has trusted access to the server public keys, he runs a configurator program that collects from the user a passphrase, the name(s) of the desired target servers, and a path to a trusted store of their public keys. The configurator then produces a `hashstore` file containing a list of server names, and (for each server) the keyed MAC, generated under a symmetric key derived from the passphrase, of the tuple of server name and public keys. The user then places this file in some publicly accessible place, such as in their home directory on the Web.

When the user then wishes to use SSH from a remote client, he initiates the connection to his home machine (e.g., `ab.cs.foo.edu`). Then, `ab.cs.foo.edu` sends its public key to the client machine. The modified SSH client prompts the user for the URL of the hashstore. The modified client fetches the hashstore and extracts the keyed MAC for `ab.cs.foo.edu`. The modified SSH client prompts the user for his passphrase. The client then derives a symmetric key from this passphrase, and uses it to generate the keyed MAC for the alleged public key. The generated MAC is compared with the MAC received from the web page. If the two values match, the client proceeds with SSH.

Variations are possible. For example, we could take the decentralization even further and, rather than having the user remember the current global name for each target host, the modified client could use the hashstore to obtain an IP address or other external name for the personal hostname the user typed (thus avoiding the `foo.bar.com` problem our colleague encountered).

Semi-Centralized In some scenarios, it might be reasonable for an enterprise to set up a CA that reliably signs certificates for SSH daemons at servers.

Our approach also permits such semi-centralized approaches. The SSH protocol will already have the server send this certificate to the client. What we need to do is provide the client with a certification path to verify this information; a minimal one might be a certificate for the user's home enterprise's root CA.

In this case, the enterprise admin set up an LDAP database, and runs a configurator tool that populates the database with a record, for each user, containing a this certification path and a keyed MAC for it, generated via the user's passphrase.

When the user then wishes to use SSH from a remote client, he initiates the connection to his home machine (e.g., `ab.cs.foo.edu`). Then, `ab.cs.foo.edu` sends its certificate to the client machine. The client prompts the user for the location of the LDAP. The client contacts the LDAP, and sends the user's name. The LDAP server looks up the username and sends back the certification path and the hashed CA certificate corresponding to that. The modified SSH client prompts the user for his passphrase. The client then derives a symmetric key from this passphrase, and uses that key to generate a keyed MAC of the certification path received. If the MACs match, the client then validates the server certificate using this certification path. The client then proceeds with SSH.

Again, many variations are possible here. For example, the CA need not coincide with the party setting up the LDAP; alternatively, an individual user could set up his Web-based hashstore (as in the decentralized approach above) to include the public keys of CAs he chooses to trust.

In this semi-centralized approach, the only role of the CA is to certify SSH daemons for its user population—which is probably not a high-volume task. Since the CA does not certify users, we anticipate that revoking and re-issuing certificates will be infrequent. Avoiding the need for a CA certificate issued by a standard trust root also reduces expense and hassle. Thus, we eliminate many potential performance and exposure issues—this machine will just not be used that often.

Furthermore, by easily allowing a user to go right to a root, this approach permits a trust hierarchy that consists of a forest of small trees. The user specifies the tree root. This eliminates the hassle of intermediate certificates (cross-certificates or bridge certificates) that would be generated if we wanted interoperability between multiple Certificate Authorities within one realm. This also provides increased resilience; compromising the security of one CA will not compromise the whole PKI.

Certificates and Expiration Since our solution's "small-delta" constraint dictated simplicity, we opted for long lifespans on server certificates and hope for minimal certificate revocation. We considered delegated path discovery and validation [6], which would enable the client to offload checking CRLs and other such duties to a remote server, but decided against it for simplicity.

In the worst case, supporting revocation of the trust root (and preventing replay attacks) would require a way to "revoke" old keyed hashes. One approach would be to have users change passphrases; another would be to have the LDAP know the users' passphrases (a risk).

We opted for X509v3 certificates. The reasons for that are multifold. First of all, the X.509 standard [1] 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.

4.3 Security

The security of the above approaches stems from the basic fact that the user can use a passphrase as a shared secret key to create a hash.

Semi-Centralized The semi-centralized approach starts by having the server `ab.cs.foo.edu` send 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 she wants to pose as `ab.cs.foo.edu`. She replaces the server certificate with her own certificate. Certificate verification would fail in such a scenario because when the SSH client looks at the uniquely identifying server name of the certificate, it would not be the one it expected. The server name would not match. If the attacker forges it to be the same, she would not be able to modify her signature to match the forged extension, as the CA generated the signature, therefore the certificate verification would fail again. The client 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 suitable certificate, even if she has a certificate issued by the same certificate authority.

The user types in the URL/LDAP address, which is connected to an LDAP server; the client sends the user name to the LDAP. 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 client uses the user passphrase to generate the hash for the modified certification path, it would not match the hash received. The attacker would not know the password used by the user. Essentially, the only way to crack this protocol is by cracking the password used. That can be done by dictionary attacks or by other techniques, such as social engineering. Therefore it is important that the user selects a long passphrase and fulfills the requirements of a good passphrase to ensure that this protocol is secure.

Revocation 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 or OCSP (e.g. [4]) can inform clients that a certificate has been revoked. In that case, the client would check whether the certificate has been revoked or not. 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 [1] 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.

4.4 Decentralized Approach

The decentralized approach starts out by having the server `ab.cs.foo.edu` sends its public key to the client machine. Suppose an attacker intercepts and replaces it with her own public key. Once the client receives the public key, it retrieves the keyed-hash of the public key. It hashes the public key and compares it with the hash received. The two hashes would not match.

Suppose the attacker now attempts to send her own keyed hash instead of the one stored at the place the user specified. To be able to generate a valid hash, the attacker needs to know the user's passphrase. Therefore if she replaces a valid hash with one of her own, that hash would not match to the one that the client would produce using the user's passphrase on the spoofed public key received.

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.foo.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 possess the public key of the server—and not the private key—implies that she 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.

Revocation Suppose the server public key changes. In such a scenario, the client would be vulnerable to a replay attack. In a simpler model, the user 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 introduces a window of risk; however, in the interest of simplicity (and in the hope that server certificates do not change often), we believe that it should suffice.

4.5 Drawbacks

Our solution does have some disadvantages.

First, our use of a hash keyed from a user-memorized passphrase introduces a risk of offline dictionary attack on the passphrase. For now, our defense here is to encourage users to use sufficiently long passphrases to minimize this risk. (Potentially, we could also augment the hash-fetch protocol to include temporary symmetric keys known only by the client and the SSH server keyholder—requiring an attacker to actively impersonate a server in order to gather material for a dictionary attack.)

Secondly, our solutions may require user participation in dealing with revocation of SSH server key pairs. Here, we believe that the relative infrequency of this activity—combined with the fact that the default scenario requires the users to actively participate in SSH key pairs anyway—will reduce the significance of this requirement.

Both of these issues remain areas for future work.

Of course, our principal disadvantage that we require that borrowed client machines have an alternate SSH client.

5 Implementation

5.1 Overview

We prototyped the fully decentralized solution above. We also have hooks in place for the semi-centralized solution (and some other techniques we discuss in Sect. 6).

The design and code that was written to develop the prototype is open source and modifies the SSH protocol.

At the time of our experiments, the existing SSH protocol supported by OpenSSH had the bare skeletal structure to support certificates that does not directly support certificate verification. OpenSSL(0.9.6c) has created routines and callback functions that can be used with much ease to verify certificates. We added a `First_Time` authentication method to carry out our decentralized approach to verifying keys. Figure 1 sketches an architectural block diagram that explains how we extended the SSH client.

For a keyed MAC, we chose HMAC [3], which is built on a cryptographic hash function, typically MD5 or SHA-1 (we chose SHA-1). We derive the symmetric key from a 20-character passphrase entered by the client.

5.2 Choice of Codebase

At the time of our experiments, open-source OpenSSH clients existed for Linux, and the only fully-deployed open source SSH server was provided by OpenSSH and and

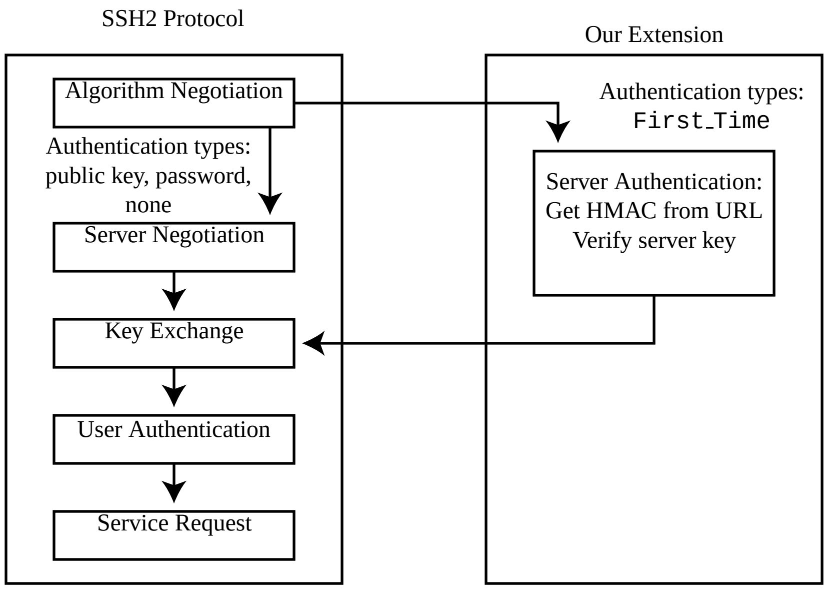

SSH2 Protocol
Algorithm Negotiation
Authentication types: public key, password, none
Server Negotiation
Key Exchange
User Authentication
Service Request
Our Extension
Authentication types: First_Time
Server Authentication: Get HMAC from URL Verify server key

Fig. 1. Architectural block diagram of the `First_Time` method.

was distributed freely with Red Hat Linux. `NetworkSimplicity.com` provides a Windows port for the OpenSSH client and server. However, using this codebase would not give us a graphical user interface.

TeraTerm SSH clients were interoperable with OpenSSH servers and provided a Graphical User Interface that makes it easy to use on Windows platform. The TeraTerm SSH client is an extension to TeraTerm Telnet client, which is also open source.

For our work, we chose the TeraTerm SSH client, primarily because we felt that a graphical Windows client would be more useful for traveling users.

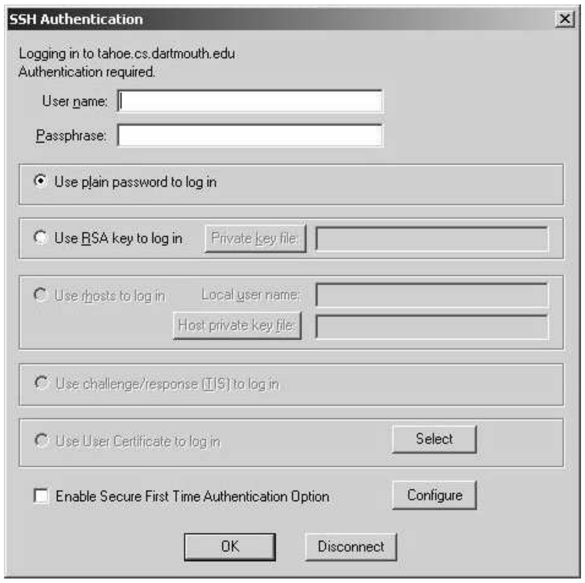
5.3 Application Interface

Figure 2 shows the main entry. To use the fully decentralized option, the user selects the “Enable Secure First Time Authentication” option and clicks “Configure.” Once the user enters the hostname he is prompted with the User Interface displayed at left in Figure 3, which prompts the user for the hashstore.

The client receives the public key/server certificate and is prompted to enter the hash password (Figure 3, right). If the password authenticates the hash, SSH authentication proceeds as normal. (Figure 4).

5.4 Configurator

Our configurator utility is an independent program that sets up the hashstore. The hashstore contains a list of the machines, that the user accesses remotely, and their corresponding Hash values.


SSH Authentication

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

(●) Use plain password to log in

Fig. 2. The main screen

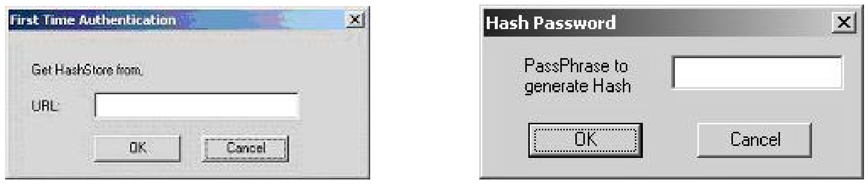
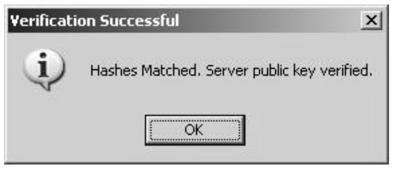


Fig. 3. At left, the client requests URL entry for hashstore; at right, the client requests the passphrase to generate the HMAC.

This program takes the server public key used for SSH sessions, concatenates it with the server name and then generates the hash and stores them in a file called `hashstore`. The file is then placed on a hosted Web site. Figure 5 shows sample of the output when this program runs.

5.5 Code Availability

All the code that was used and modified in this project is open source and can be downloaded from `http://www.cs.dartmouth.edu/˜sws/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.


Verification Successful

Hashes Matched. Server public key verified.

Fig. 4. If the retrieved HMAC matches the public key the server sent, then the server has been authenticated.

```
 $./linuxConf
 This program generates the HMAC Hashes for the public
keys of host machines. This program is to be run on the local
machine whose public key is to be hashed

Enter the host name of the machine: foo.bar.com
----String to be Hashed---
foo.bar.com 1024 35
151052055305049513313873325220639580170308824866279975903865
17065659085280834883625163014772300422177712761876894150979694040698019214365467
07381986625813796034665095626099575794106722934274328194857789445170395185623181
06109502073419576490304279566550418746874044392631514426610329187514765207346921
326949181
Enter the pass phrase to hash:
MySecretPassPhrase%1

---Hash stored in hashstore.txt---
foo.bar.com 1024 35 MD:_YłS (non ascii characters not displayed)
$
```

Fig. 5. Sample output of the configurator, which generates HMAC values of the public keys, to server as the user's trust roots from remote locations. (The use of the keyed MAC ensures integrity without a universal PKI.)

6 Conclusions and Future Work

Our solution brings public-key security to SSH in a a flexible and scalable way—and (when one considers the decentralized approach) constitutes a datapoint for a useful "PKI" with neither CA nor certificate.

The ease of use and deployment of the "decentralized non-CA approach" distinguishes our solution from one based on a traditional hierarchical PKI.

Currently, both academia and industry strive to develop standardized protocols that would make the deployment of PKI relatively manageable; however, a look at the deployed technology base shows that we're not quite there yet. Most small enterprises do not need to invest and develop a hierarchical trust model for a PKI. 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 set up a PKI. In contrast, our centralized CA approach is suited for larger networks with several users. It develops a PKI with a minimal set of components.

In future work, we would like to take the prototype to another level of completion. Right now, the user interface is not completely clear; and arguably, a secure client for traveling users should be aggressive about *not* retaining validated server keys. We would also like to extend the code to handle more pieces of the solution space; for example, the current code does not support the semi-centralized approach (beyond connecting to an LDAP server). We also would like to strengthen the way that the HMAC keys are derived from passphrases. Developing a Linux version would also be useful. Exploring *visual hashing* [5] as an alternative way for a human user to authenticate complex data would also be interesting.

References

1. Carlisle Adams and Stephen Farrell. “Internet X.509 Public Key Infrastructure Certificate Management Protocols.” IETF RFC 2510, March 1999.
2. Daniel J. Barrett and Richard E. Silverman. *SSH: The Secure Shell, The Definitive Guide.* O’Reilly & Associates. 2001.
3. H. Krawczyk, M. Bellare, R. Canetti. “HMAC: Keyed Hashing for Message Authentication.” RFC 2104, February 1997.
4. Michael Myers, Rich Ankney, Carlisle Adams, Stephen Farrell, and Carlin Covey. “Online Certificate Status Protocol, version 2.” Internet Draft, March 2001.
5. Adrian Perrig and Dawn Sogn. “Hash Visualization: A New Technique to Improve Real-World Security.” *International Workshop on Cryptographic Techniques and E-Commerce*. 1999
6. Denis Pinkas, Russ Housley. “Delegated Path Validation and Delegated Path Discovery Protocol Requirements.” Internet Draft, February, 2002.
7. J. Schlyter, W. Griffin. “Using DNS to Securely Publish SSH Key Fingerprints.” Secure Shell Working Group, Internet Draft. September 2003.
8. Dawng Song, David Wagner, Xuqing Tian. “Timing Analysis of Keystrokes and Timing Attacks on SSH.” *10th USENIX Security Symposium*. 2001.
9. Siva Sai Yerubandi, Weetit Wanalertlak. “SSH1 Man in the Middle Attack.” Oregon State University. `http://islab.oregonstate.edu/koc/ece478/project/2002RP/YW.pdf`. 2002.
10. T. Ylonen, D. Moffat. “SSH Protocol Architecture.” Network Working group, Internet Draft, Octber 2003.
11. T. Ylonen, D. Moffat. “SSH Connection Protocol.” Network Working group, Internet Draft, October 2003.
12. T. Ylonen, D. Moffat. “SSH Transport Layer Protocol.” Network Working group, Internet Draft, October 2003.
13. T. Ylonen, D. Moffat. “SSH Authentication Protocol.” Network Working group, Internet Draft, September 2002.