SECURE LOGIN OVER TCP/IP USING PUBLIC-KEY CRYPTOSYSTEM

By

PASSAKON PRATHOMBUTR

Master of Science

in Computer Science

Chulalongkorn University

Bangkok, Thailand

1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1998

SECURE LOGIN OVER TCP/IP USING

PUBLIC-KEY CRYPTOSYSTEM

Thesis Approved:

H. Lu

Thesis Advisor

J Chandler

[illegible]

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGEMENTS

I am indebted to my loving wife Cholticha for her understanding and her support throughout my work on this. Her support is constant and strong. Without it I would not succeed. I would like to express my sincere appreciation to my thesis advisor Dr. Huizhu Lu who spent her valuable time to advice and support me. Also, I would like to extend my thanks to Dr. J. Chandler and Dr. H. K. Dai for their advice and willingness to serve on my graduate committee. Their suggestions and support were very helpful throughout the study. Finally, I would like to express my gratitude to my parents who were mentally with me all the time.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The authorization of the Unix user depends on his/her password. The password is mostly secure whenever the user accesses his/her server on a private network. Nonetheless, whenever a user accesses the Unix server from machine outside the office across unprotected networks like the Internet, how can the user protect his/her password over the Internet network? Nowadays the usage of passwords over the Internet is a major security problem. For example, early in 1994, thousands of passwords were trapped by the sniffer programs installed in various university networks connected to the Internet. At the end of that year, a number of attacks were successfully launched by Kevin Mitnick against several computer centers, including the San Diego Center for Supercomputing [Shimomura 1996].

The Internet is an open network based on the Transmission Control Protocol / Internet Protocol (TCP/IP) stack. By the nature of TCP/IP, it was not designed to secure the data at all. No encryption was provided in it. The data flowing on the TCP/IP network could be trapped by the man-in-the-middle. However, several technologies and algorithms have been applied to secure the messages carried over the TCP/IP network, i.e., Secure Sockets Layer (SSL) for a web page, Secure Shell (SSH) for a Unix shell and Pretty Good Privacy (PGP) for email. In the case of a secure login, there are at least two techniques of one-time password (OTP) operating on the Internet, the S/KEY and the

RFC-1938. The OTP behaves exactly as its name indicates: a user has to change his/her password each time it is used. The disadvantage of using the OTP is that a user needs to keep the preinstalled passwords or OTP calculator with him/her. The user is probably no longer using his/her original Unix password even though he/she logs in at a console.

How can a user remain using his original password over an insecure network? The features of public key encryption and Java programming could solve this problem. With a basic concept of public key encryption, a user can easily encrypt his/her password by a common Java Web page on any platform before sending it to verify the logging authorization at the server. Obviously, the user requires neither specific programs nor materials with him/her. No insecure messages are stored on the user part and the network; this implies an eavesdropping protection. It is possible to design this method as an alternative for the secure login techniques.

The objective of this thesis is to design and implement a secure login system for a Unix operating system using an encryption algorithm to secure a password over the TCP/IP network. The designed system must be convenient for the user to log in from any hosts and any platforms without any modification or configuration. The user's machine should require only a web browser which is usually available on the TCP/IP client computer. Also, the user is able to retain his/her original Unix password rather than the list of one-time passwords.

To achieve the objective, the proposed technique decides on the RSA public-key cryptography that can be applied to an educational application program. Furthermore, the RSA algorithm is widely used for authentication and encryption in the computer industry today. The RSA technique uses a pair of asymmetric keys, a public key and a private key, for encryption and decryption. The ciphertext or the encrypted message using the public key is merely decrypted by the private key and vice versa. The public key can be made public, whereas the private key must be kept in secret. The diagram in Figure 1.1 shows the basic concept of asymmetric key encryption.

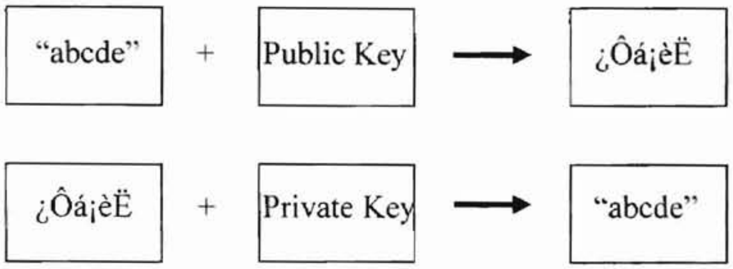


Figure 1.1 The diagram of asymmetric key encryption.

In the secure login program, the password should be encrypted by the RSA public key on the user machine before delivering to the server machine. The encryption program could be implemented in the form of a Java applet retrieved from web browser. Therefore, a user can access the secure login system from any computer platform. Obviously, eavesdropping is useless because the password was encrypted and the public key cannot decrypt the password. The private key that can decrypt the password is kept secretly on the server. Only the privileged programs on the server can decrypt the password and grant login. With these properties, the secure login system was designed and implemented.

CHAPTER II

LITERATURE REVIEW

2.1 Cryptosystems

A cryptosystem is a system to encrypt/decrypt these two forms of messages: a message known as "plaintext" and another message known as "ciphertext", using a mathematical function and a special password called the "key". The cryptosystem is an important aspect of computer network security and is becoming increasingly important as the tool of electronic commerce. It protects the valuable information flowing within a network. Many cryptography techniques have been invented. Encryption can be classified into two common techniques: conventional or symmetric encryption and public-key or asymmetric encryption.

In conventional encryption, plaintext is converted to apparently random nonsense referred to as ciphertext, by using one key. The key is a value independent of the plaintext. Changing the key changes an output or a ciphertext. Once the ciphertext is created and transmitted to a receiver, the receiver must apply the same key to decrypt the ciphertext. The secrecy of the conventional encryption depends on the security of the key itself, not the secrecy of the algorithm. The key length is a major design issue for the encryption algorithm because the longer the key, the higher the work factor the cryptanalyst has to crack. Other problems are how to deliver and how to secure the key.

In public-key encryption, the plaintext is always encrypted and decrypted using different keys, with the required characteristic that one cannot be derived from the other and those two keys must be matched. The public-key encryption algorithm can apply to various applications such as a digital signature, a message encryption, and a key exchange.

The best-known conventional encryptions are the International Data Encryption Algorithm (IDEA) and the Data Encryption Standard (DES). The best-known public-key method is the RSA.

IDEA: IDEA was developed by Xuejia Lai and James Massey of the Swiss Federal Institute of Technology. It is a block cipher that uses a 128-bit key to encrypt data in blocks of 64 bits. No currently known technique is able to break IDEA. IDEA has been proposed as an international encryption standard in recent years [Hughes 1995].

DES: DES was developed by IBM and was adopted by the National Bureau of Standards in 1977. Nowadays it has become the standard of the National Institute of Standards and Technology and is also known as the Federal Information Processing Standard 46 (FIPS PUB 46). DES uses a 56-bit key to encrypt data in 64-bit blocks [Stallings1995].

Figure 2.1 displays the chart of encryption algorithms and their related applications. Notice that most applications usually require a combination of algorithms. For example, PGP requires MD5 to digest a mail message, DES to encrypt the message, and RSA to encrypt the DES key. The applications illustrated in Figure 2.1 are:

PGP: Pretty Good Privacy is a program used to exchange encrypted and authenticated email messages. PGP was invented by Philip Zimmermann in 1991. It provides two services: encryption and authentication by means of a digital signature. PGP encrypts the message using IDEA with a 128-bit pseudo-random key. To secure the key exchange, this key is then encrypted by RSA using the recipient's public key. Only the recipient can recover the IDEA key and only this IDEA key can decrypt the message. PGP creates a digital signature using MD5 and RSA. MD5 digests the mail body into a fixed-length message. Then RSA signs it by the sender's private key.

Today PGP versions exist for almost all major computer platforms, including Windows 95, Windows NT, OS/2, MacOS, Unix and VMS.

PEM: Privacy Enhanced Mail, as documented in RFCs 1421 through 1424, is a standard for message encryption and the authentication of message senders. PEM creates a digital signature using MD5 and RSA. This methodology provides for authentication, integrity, and non-repudiation of the original message. PEM can also encrypt the message using DES and RSA. (It is simply not suitable to use RSA alone, because RSA operations are on the order of one hundred times slower than DES.) Unlike PGP, PEM uses a pseudo-random 56-bit DES key to encrypt the message. Then this key is encrypted by RSA using the recipient's public key. The ciphertext and encrypted secret key (DES) are sent to the recipient. Obviously, only the recipient's private RSA key can decrypt the secret key, and only the secret key can decrypt the ciphertext.

Today, several PEM-compliant programs exist in various computer platforms including Unix, OS/2, MacOS and Windows NT.

SNMPv2: Simple Network Management Protocol version 2 is a protocol used to monitor and control the activities and behavior of devices on the TCP/IP network. The SNMP enables a management station to configure network devices. The SNMP Version 2 supports several viable security mechanisms described in RFC 1446. The SNMPv2 provides authentication and confidentiality services using DES and MD5 [Hughes 1995].

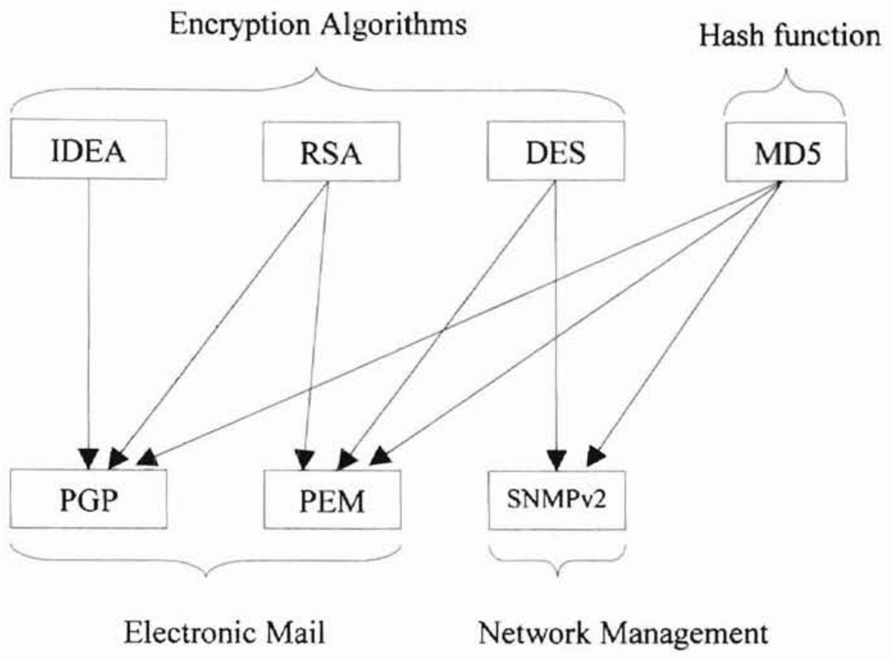


Figure 2.1. The encryption algorithms and their related applications.

2.2 RSA Algorithm

RSA is a public key cryptography which can be applied for an authentication. The RSA algorithm was developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. At the present, the RSA algorithm is the most commonly used in encryption and authentication and is included as part of many commercial products, such as Netscape, Quicken, and Lotus Notes. Moreover, it is proposed to many standard organizations, e.g. ITU-T, ISO, ANSI and IEEE [http://www.rsa.com/rsalabs/newfaq/q8.html].

2.2.1 The correctness of RSA.

Given the message M and the ciphertext C, the RSA encryption and decryption algorithms are performed by (1) and (2), respectively [Feil 1996].

$$C = M^e \bmod n \qquad (1)$$

$$M = C^d \bmod n \qquad (2)$$

Where the public key is the set of two integers $\{e, n\}$ and the private key is the set of two integers $\{d, n\}$.

How can we calculate the value of e, d and n to be satisfactory? Begin with the Euler's Totient Function ($\Phi(n)$); it is the number of positive integers less than n and relatively prime to n, for any positive integer n. (*Note:* two numbers are said to be relatively prime or coprime to n if their greatest common divisor, gcd, is 1). For example, $\Phi(10) = 4$, since 1, 3, 7 and 9 are less than 10 and relatively prime to 10. For a prime number p, $\Phi(p) = p - 1$ and for two distinct prime numbers p and q, $\Phi(pq) = (p - 1)(q - 1)$.

Given two integers n and M, and two distinct primes p and q such that M is relatively prime to n, $n = pq$ and $0 < M < n$, the following relationship, known as the Euler's theorem, holds:

$$M^{\Phi(n)} = M^{(p-1)(q-1)} \equiv 1 \bmod n$$

or

$$n \mid (M^{\Phi(n)} - 1)$$

or

$$M^{\Phi(n)} \bmod n = 1.$$

Thus $M^{k\Phi(n)} \bmod n = 1^k$ for any integer k and

$$M^{k\Phi(n)+1} \bmod n = M.1^k = M \qquad (3)$$

If p and q are big primes, $\Phi(n)$ depends on being able to factor n, which is computationally impractical.

Given two positive integers e and d, which are multiplicative inverses, mod $\Phi(n)$ i.e., $ed \bmod \Phi(n) = 1$.

Thus, there exists an integer k such that:

$$ed = k\Phi(n) + 1 \qquad (4)$$

The M is then recovered by:

$$\begin{aligned} C^d \bmod n &= M^{ed} \bmod n && \textit{replace C from (1) into (2)} \\ &= M^{k\Phi(n)+1} \bmod n && \textit{replace ed from (4)} \\ &= M && \textit{by (3)} \end{aligned}$$

2.2.2 The use of RSA

The use of RSA is classified into three steps:

1. Private key and public key creation

2. Encrypt a message

3. Decrypt or verify a message.

1. Private key and public key creation

The initial step is to generate a pair of keys. This step is usually performed only once unless the keys are lost. The key in this term is a big number performed like a secret code or password. To generate the pair of keys, select two large primes, p and q, then compute the modulus n:

$$n = pq.$$

The modulus n is the product of two large primes, so it could not be easily factored. Next, compute the $\Phi(n)$:

$$\Phi(n) = (p\text{-}1)(q\text{-}1).$$

This number represents the quantity of numbers less than or equal to n that are relatively prime to n.

Finally, select some number e that is relatively prime to $\Phi(n)$ and find another number d such that:

$$ed \bmod \Phi(n) = 1$$

The pair (n,e) is the key that can be made public. While the pair (n,d) is the private key, which must be kept in secret. The p, q, and n may be saved with the private key or destroyed, because they could be used to calculate d.

2. Encrypt a message

A message M is encrypted by using the private key of a sender with the following exponentiating:

$$C = M^d \bmod n$$

Where d and n are the private key of the sender.

The encrypted message or ciphertext C is now safe to send out.

3. Decrypt or verify a message

When the recipient gets the ciphertext C, the message M can be decrypted from C by the public key of the sender with the following exponentiating:

$$M = C^e \bmod n$$

Where e and n are the public key of the sender.

The message M in the above equation can be represented by the ASCII codes of a password or a digested message generated by a message digest (MD) algorithm. In case of digital signature, since the mail body is too large for RSA encryption, it must be first digested and then encrypted by the sender's private key to form a digital signature. For example, PGP applies MD5 to digest a mail and encrypt its result by RSA to form the digital signature.

Example: Generate the two keys of RSA

First, select two primes: $p = 7$, $q = 17$ (this is a simplified version; in fact, p and q are much bigger than this).

$$n = pq$$

$$n = 119$$

$$\Phi(n) = (p-1)(q-1) = 96$$

Next, select $e = 5$ because 5 is relatively prime to 96.

By $\quad ed \bmod [(p-1)(q-1)] = 1$

So $\quad d = 77$

The public key is $e = 5$ and $n = 119$.

The private key is $d = 77$ and $n = 119$.

Suppose that a message M is a number 19. The creation of encrypted message shown below.

$$C = M^e \bmod n$$

$$C = 19^5 \bmod 119$$

$$C = 66$$

The encrypted C is sent to the recipient. When recipient receives the encrypted C, the original message is decrypted by the following formula:

$$M = C^d \bmod n$$

$$M = 66^{77} \bmod 119$$

$$M = 19$$

Finally, the message is recovered.

Although the RSA technique provides an attractive feature of authentication, it requires rather intensive mathematical computations. Fortunately, there are mathematical techniques illustrated in Chapter III that reduce the computation. Moreover, CPU technology grows quickly and the computational time could be ignorable at some time in the future.

2.3 The Password Schemes

In a computer network, especially public network, a user's password transmitted in clear text represents a major vulnerability. There are several methods to capture and crack the password in the network, e.g. spoofing (one entity pretends to be a different entity to gain unauthorized access). In general, attacks by an intruder occur in two manners, passive and active attacks. By the manner of the passive attack, the intruder in the middle observes the data transmitted between sender and receiver. For example, the LAN analyzer can capture the TCP frame on the local area network. Thus the passwords packed into the TCP frame could be trapped. By the manner of the active attack, the intruder is able to interrupt, modify or fabricate the data as shown in Figure 2.2. The intruder can also redirect or delay traffic to make the system malfunction. Hence, the developer of the authentication program has to consider every possibility of attack [Oppliger 1998]. Nevertheless, the perfect design does not guarantee absolute prevention. The system administrator has to lookout for an intruder and maintains the system to avoid the possible attack, e.g. patch or upgrade the kernel, which fixes the known security holes.

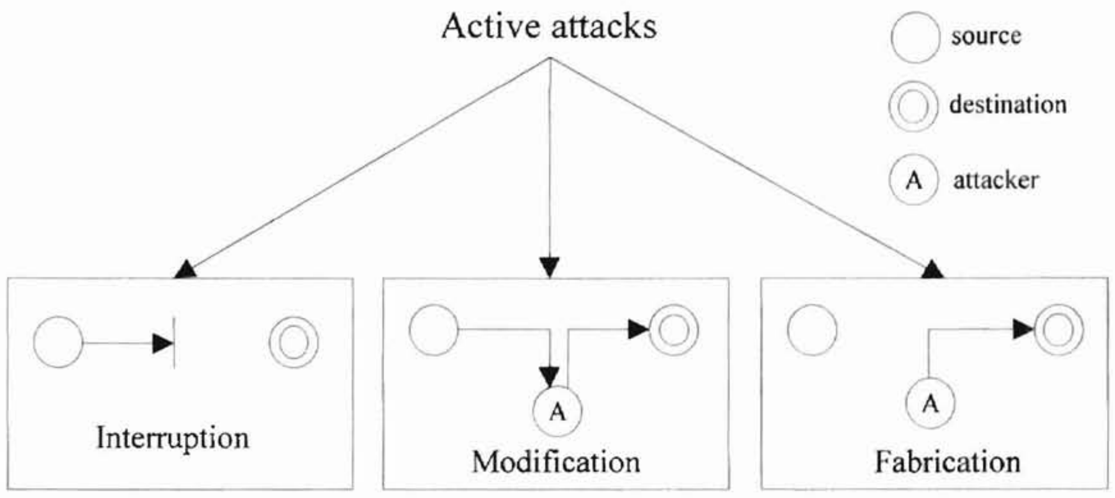


Figure 2.2. Three types of active attacks.

A password, associated with each user, is typically a string of six to ten or more characters that a user is capable of committing to memory. The password is the primitive way to gain authorized access to a resource over the network. Obviously, it is insecure for a user to enter his/her password over the network. However, various passwords and encryption schemes are proposed to protect the using of passwords over the network. A subsequent section reviews the existing schemes.

2.3.1 One-time password

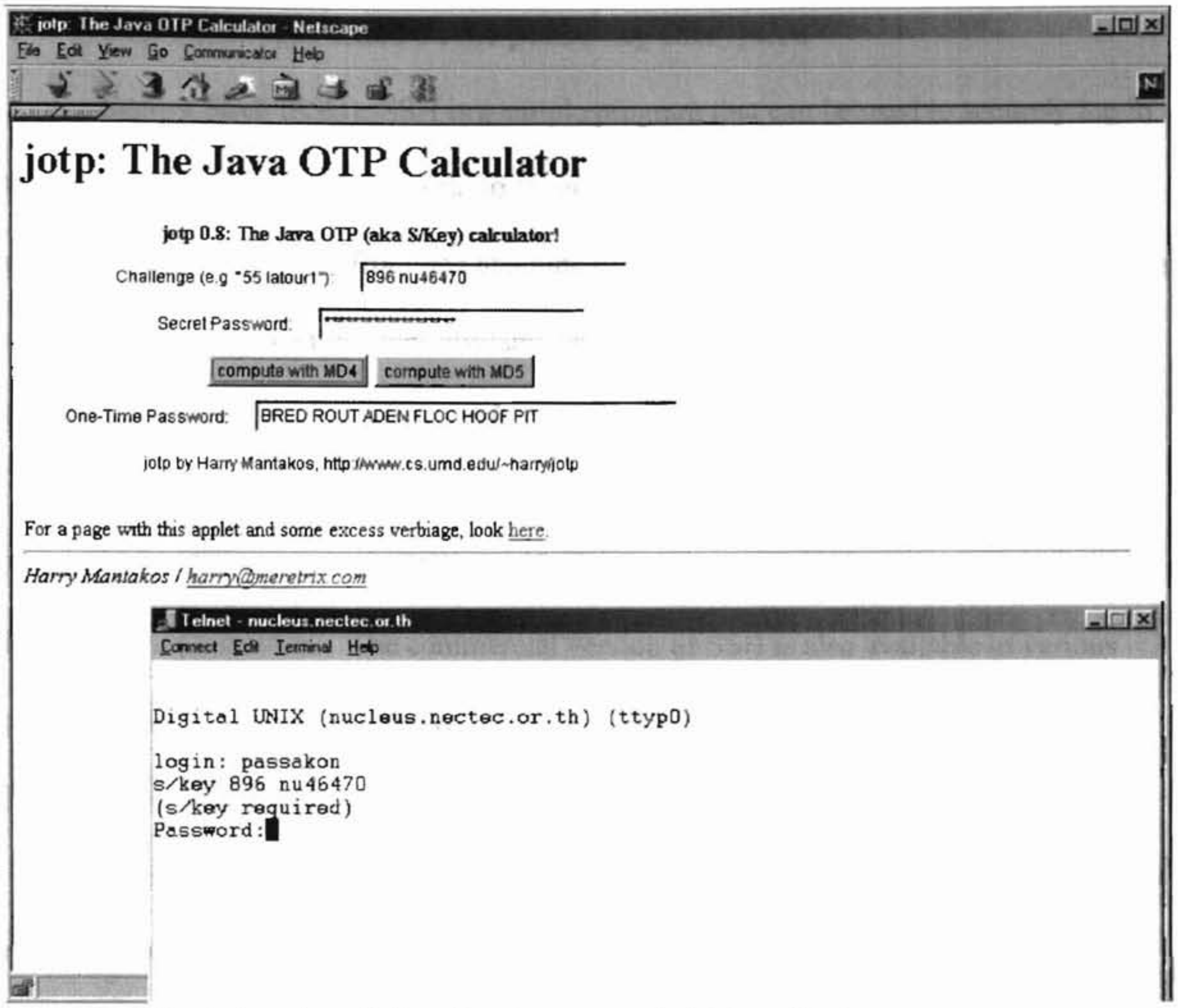
The one-time password is a password that is used only once. Therefore, a user probably requires a list of passwords or a password calculator in hand to generate the next password, in hand. Obviously, with the one-time password, eavesdropping the password is worthless. Plenty of one-time password concepts have been introduced. Three of them excerpted from [Menezes 1996] include:

1. *Shared lists of one-time passwords*. The user and the system use a sequence or set of secret passwords, each valid for a single authentication, distributed as a pre-shared list.
2. *Sequentially updated one-time passwords*. Initially only a single secret password is shared. During authentication using password *i*, the user creates and transmits to the system a new password (password $i + 1$) encrypted under a key derived from password *i*. This method becomes difficult if communication failures occur.
3. *One-time password sequences based on a one-way function*. The user begins with a secret w. A one-way function H is used to define the password sequence: $w, H(w), H(H(w)), \ldots, H^t(w)$. This method is more efficient than sequentially updated one-time passwords in terms of bandwidth.

The S/KEY: The S/KEY is a one-time password sequence based on a one-way function (hash function). It is also known as RFC-1760, the recommended password scheme for use on the Internet. The authentication of the S/KEY relies on a secure hash function called the Message Digest 4 (MD4). A secure hash function is a one-way function that is easy to compute in a forward direction but computationally infeasible to invert. From this feature, a sequence of one-time passwords is generated. The passwords are related in a way that makes it computationally intractable to compute any password from the preceding sequence. It is simple to compute previous passwords from the current one [http://www.nic.surfnet.nl/surfnet/projects/surf-ace/mm-lab/security /skey.html]. Figure 2.3 presents the usage of S/KEY to login to a Unix system.

The user requires a telnet program and a web browser program linked to the Java OTP Calculator (the jotp applet). The jotp applet requires two inputs to calculate the one-time password: i.e., the challenge and the user's secret password. On the telnet program, once a user enters his/her loginname, the server will return the challenge corresponding to the user (see the telnet window in Figure 2.3). The challenge is a combination of a sequence and a seed. The sequence is the number of loops to perform the MD4 function. Some users may apply only one secret password on multiple servers, so the seed is required to distinguish the result or the S/KEY. The jotp applet generates the S/KEY by concatenating the seed to the secret password and then performs the MD4 for the "sequence" times. The result of MD4 will be translated into the S/KEY in terms of six English words by looking up the predefined dictionary table. The S/KEY illustrated in Figure 2.3 is "BRED ROUT ADEN FLOC HOOF PIT." It was generated by using "896" as a sequence and "nu46470" as a seed. That means the MD4 has been applied to the user's secret password and seed for exactly 896 times. To login, the user has to copy the S/KEY into the password prompt in the telnet program.

Since the server already has the password of the sequence "897" stored secretly, the server can easily verify the user's password by applying the MD4 function on the received password one more time and comparing the result with the password of the sequence "897." If they match, the password of the sequence "896" is stored (for using in the next login) and the user is able to login. For the next login, the server will return the challenge for this user with the value of "895 nu46470." An eavesdropper cannot generate the password of the sequence "895" or the lower sequence because doing so would require inverting the hash function.


jotp: The Java OTP Calculator - Netscape
File Edit View Go Communicator Help

jotp: The Java OTP Calculator

jotp 0.8: The Java OTP (aka S/Key) calculator!

Challenge (e.g "55 latour1"): 896 nu46470

Secret Password: ****************

compute with MD4
compute with MD5

One-Time Password: BRED ROUT ADEN FLOC HOOF PIT

jotp by Harry Mantakos, http://www.cs.umd.edu/~harry/jotp

For a page with this applet and some excess verbiage, look here.

Harry Mantakos / harry@meretrix.com

Telnet - nucleus.nectec.or.th
Connect Edit Terminal Help

```
Digital UNIX (nucleus.nectec.or.th) (ttyp0)

login: passakon
s/key 896 nu46470
(s/key required)
Password:
```

Note: The OTP calculator is available at http://www.cs.umd.edu/~harry/jotp/.

Figure 2.3. The Java OTP Calculator and the telnet program.

2.3.2 Passwords based on the public key encryption

Secure Shell (SSH): SSH is a simple program that can be used to securely log in to another computer over a network, to securely execute commands in a remote machine, and to securely move files from one machine to another. It provides strong authentication and secure communications over insecure channels. It is intended as a replacement for the Berkeley r-tools such as *rlogin, rsh*, *rcp* and *rdist* on both client and server machines.

SSH was developed by Tatu Ylonen from the Helsinki University of Technology, Finland. The free version of SSH is available at http://www.ssh.fi/ or http://www.cs.hut.fi/ssh/. The commercial version of SSH is also available in various platforms, such as Windows 95, Windows NT, OS/2, and MacOS [http://www.DataFellows.com].

SSH normally listens for connections on TCP port 22. This port number has been registered with the Internet Assigned Numbers Authority (IANA) and has been officially assigned for SSH [http://www.ssh.net]. SSH provides the user authentication, data compression, data confidentiality, and integrity protection.

SSH starts with the client computer request to secure connection to the server as illustrated in Figure 2.4. The server sends the public host key, which is typically a 1,024-bit RSA key, and the public server key, which is typically a 768-bit RSA key that changes every hour by default. Since the RSA public key operations consume computational time, it is not a smart idea to encrypt/decrypt every transaction using RSA. Thus the public host key and the public server key are used only at the beginning. The public server key that is changed periodically has never been saved on disk, so that it reinforces the use of a public host key. The purpose of the public host key and the public server key is to encrypt the session key used to secure the transactions between the client and the server. The conventional encryption using a session key takes less computational time than public-key encryption. The client can select one of these conventional keys as a session key i.e., Blowfish, DES or Triple-DES.

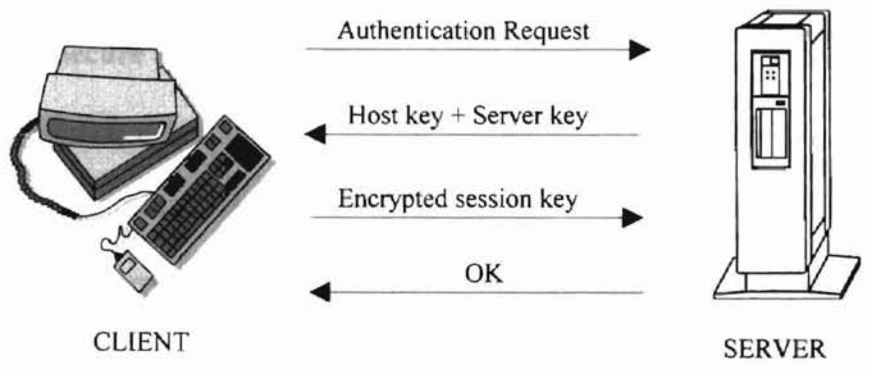


Figure 2.4. The SSH protocol execution.

Once the client receives the public host key and public server key, it may reject or accept these keys depending on the policy configuration. If the client accepts the keys, it generates a 256-bit random number to serve as a session key. The client pads the session key with random bytes, and double encrypts it with the public host key and the public server key, respectively. The result is sent back to the server. The server, in turn, decrypts the ciphertext and recovers the session key accordingly. The server sends an encrypted confirmation using a session key to the client. If the client receipt of this confirmation is successful, it implies that both parties can now start using the session key and transparently encrypt the connection.

In the case of the password scheme, the password is typically passed to the server using the session key encryption [Oppliger 1998].

The drawback of SSH is its restriction on the installation. Both client and server machines are required to install the SSH program. It is inconvenient for the user who logs in from a public machine.

Secure Sockets Layer (SSL): SSL is a program layer created by Netscape, Inc. for managing the security of message transmissions on a network. The "sockets," a part of the term "SSL," refers to the socket method of passing data back and forth between a client and a server program on a network or between program layers on the same computer. Netscape's SSL uses RSA public key encryption, which also includes the use of a digital certificate.

The SSL is layered between the application layer and the TCP layer illustrated in Figure 2.5. Thus, it could be applied by any TCP/IP applications. The SSL consists of two subprotocols, namely the SSL record protocol and the SSL handshake protocol located in order as illustrated in Figure 2.5. The SSL record protocol provides data authenticity, confidentiality, and integrity services, as well as replay protection over a connection-oriented reliable transport service. Several SSL protocols may be layered above the record protocol. The SSL handshake protocol provides an authentication and key exchange. Negotiating to initialize and synchronizing security parameters at both peers are performed by the SSL handshake protocol. After the SSL handshake protocol completes, sensitive application data may be sent via the SSL record protocol according to the negotated security parameters.

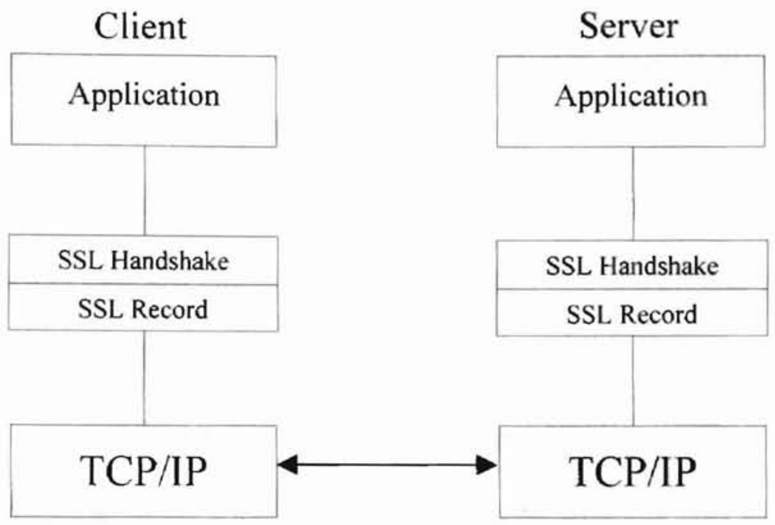


Figure 2.5. The architectural placement of SSL.

Netscape includes the client part of the SSL as a part of the Netscape web browser. If a web site is on a Netscape server, the SSL can be enabled and the specific web pages can be identified as requiring SSL access. Other servers can be enabled by using Netscape's SSLRef program library, which can be downloaded for non-commercial use or licensed for commercial use.

Netscape has offered SSL as a proposed standard protocol to the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) as a standard security approach for Web browsers and servers. Moreover, The SSL supports several applications including the *telnet* protocol at TCP port 992, but they are not yet assigned officially by the IANA [Oppliger 1998].

2.4 Unix Login System over the TCP/IP Network

Every Unix user has an account identified by a username and a password associated with it. The password is used to protect the user account from attackers, as well as to verify the user authorization. Generally, a password is a clear text between six to ten characters in length; e.g., the password of the Unix system V release 4 must be at least six characters long and contain at least two letters and one digit or punctuation character. It cannot be the same as the username, and the new password must have at least three characters different from the old one [Curry 1992].

The passwords in the Unix system are encrypted by a one-way function called the *crypt* function and stored in the standard password file. In general, Unix passwords are stored in the */etc/passwd* file. Figure 2.6 shows an example of the */etc/passwd* file which contains the username, encrypted password, real name and user's shell. Note: To hide the encrypted password from the user, some systems store the encrypted password in another file, namely the shadow password file. Minimally this file contains only the username and the encrypted password.

```
john:AqhviHRWKiMJY:502:100:John Anderson:/home/john:/bin/bash
boby:d5SHsa.xekh1Y:503:100:Boby Earle:/home/boby:/bin/tcsh
```

$\underbrace{\texttt{d5SHsa.xekh1Y}}_{\text{Encrypted password}}$

Figure 2.6. The example of the */etc/passwd* file.

The */etc/passwd* file can be read and modified only by the privileged programs. For example, the login program reads the username and the encrypted password to check for the user's authorization. When a user logs in, the login program reads the password that the user typed in and calls the *crypt* function to encrypt a 64-bit block of zeros by using the user's password as a key. The 64-bit block of ciphertext is then re-encrypted with the user's password for a total of 25 times. The final ciphertext is unpacked into a string of eleven printable characters, known as the encrypted password. Next, the login program compares the result with the encrypted password stored in the */etc/passwd* file. If the two texts match, the system allows the user to login [Garfinkel 1996].

Securing access to a system on the network is as important as securing the system itself. To log into the Unix system over the TCP/IP network, the user has to startup the login client program known as the *telnet* program. The *telnet* is a remote terminal service running on TCP port 23.

The TCP ports are identified by nonnegative integers in the range of 0 to 65,535. The best-known TCP port numbers are the first 1,024 ports, which are managed and assigned by the Internet Assigned Numbers Authority. Each TCP port has the service program corresponding to it. On the Unix system, the assigned TCP ports are stored in the */etc/services* file. Each line of the */etc/services* file consists of a service name, a TCP port number, a protocol name and a list of aliases. Figure 2.7 shows the example of services in the */etc/services* file.

```
ftp             21/tcp          file transfer protocol
telnet          23/tcp          remote pseudo terminal
smtp            25/tcp          mail
```

Figure 2.7. The example of services in the */etc/services* file.

Most versions of UNIX that support the Berkeley networking utilities have the *inetd* (the Internet daemon or the Internet superserver) to start many service programs that provide network services. The *inetd* waits for network connections on a number of the TCP ports, and when the request for the connection of specific port arrives, it invokes the services program that should handle that request. The configuration of *inetd* is stored at */etc/inetd.conf* file. The parameters in the */etc/inetd.conf* file are the name of service, the type of connection, the protocol that service expects to use, (no) wait options, the user id that should run the service program, the path name of the service program and the arguments. Figure 2.8 displays the content of the */etc/inetd.conf* file.

```
ftp     stream  tcp  nowait  root  /local/etc/tcpd        in.ftpd
telnet  stream  tcp  nowait  root  /local/etc/tcpd        in.telnetd
name    dgram   udp  wait    root  /usr/sbin/in.tnamed  in.tnamed
shell   stream  tcp  nowait  root  /local/etc/tcpd        in.rshd
login   stream  tcp  nowait  root  /local/etc/tcpd        in.rlogind
exec    stream  tcp  nowait  root  /usr/sbin/in.rexecd  in.rexecd
talk    dgram   udp  wait    root  /usr/sbin/in.talkd   in.talkd
```

Figure 2.8 The example of the */etc/inetd.conf* file.

The *telnetd* is the service program sleeping in the background, waiting for the *telnet* connection. Once there is a request from the client machine to the TCP port 23 of the server, the *inetd* will invoke the *telnetd* program to handle the *telnet* service. The *telnetd* process then opens the pseudo terminal for the *telnet* client and splits into two processes using *fork*. The parent process maintains the communication across the TCP connection, and the child does an exec of the *login* program. The *login* program begins checking the user's authorization by asking for the username, calling the *getpwnam* system-call to fetch the user password file entry, and calling the *getpass* function to display the "Password:" prompt and to read the user's password thereafter.

The login program then encrypts the password using the *crypt* function and compares the result with the encrypted password from the password file entry. If the login attempt fails, the login process will terminate or start the *login* program over again for a limited number of times. If the login is correct, the *login* program will setup the Unix environment, like the home directory, shell, username, and path, for the user. The access permissions are also changed for the terminal device, so that user-read, user-write and group-read are enabled for the user. Finally the *login* program sets the user-id and invokes the user's shell [Stevens 1993]. Figure 2.9 shows the sequence of processes that are involved in executing the *telnetd*.

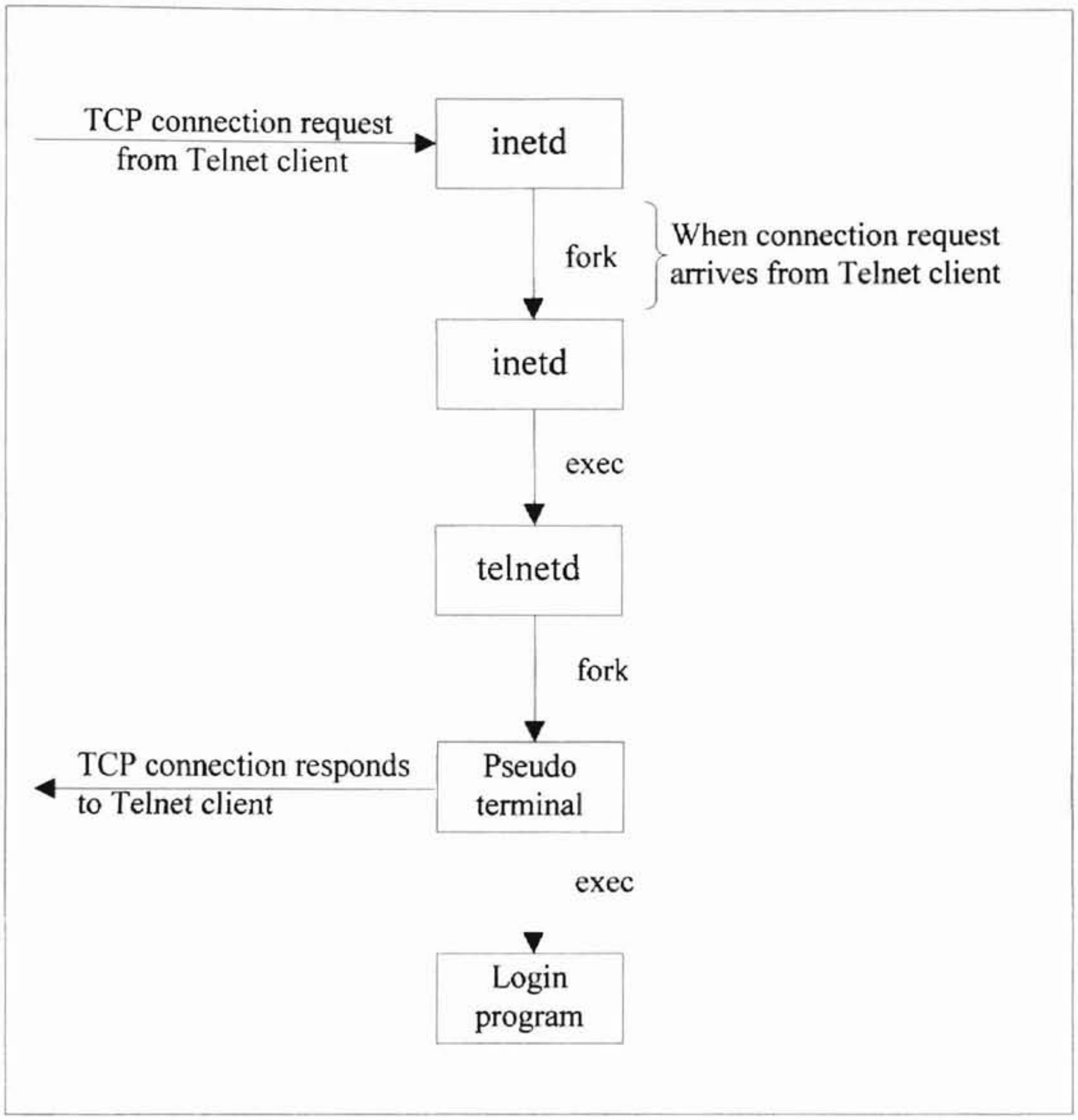

TCP connection request from Telnet client
inetd
fork
When connection request arrives from Telnet client
inetd
exec
telnetd
fork
TCP connection responds to Telnet client
Pseudo terminal
exec
Login program

Figure 2.9. Sequence of the remote login process.

CHAPTER III

DESIGN AND IMPLEMENTATION

3.1 Design

The design of computer network security applications has to consider the security services. The secure login program considers the security services in the following subsection.

3.1.1 Security services

Confidentiality: Private information is accessible only by the authorized user. On the Unix system, each user must have a unique user-id related to the username/password. This user-id is used by the Unix system to determine access rights to various files and services. Only the *root* or *superuser* has the right over every user on the Unix system. The *root* is a special user with user-id of 0 in the */etc/passwd* file. The Unix system uses the *root* to accomplish the system administrator jobs [Garfinkle 1996].

The secure login program designs confidentiality based on the authorization of the Unix system by using the *root*'s right to manage privacy information and the authorization for the Unix login system.

Authentication: Authentication is proving the genuineness of the user. Assume that a username and a password are unique for each user. The user could identify himself/herself by the password. The secure login program protects the passwords that flow between clients and servers by the encryption algorithm.

Access control: Access control is the ability to limit and control access to host systems and applications via communications links. The secure login program itself does not provide the access control function, but it could be applied to any access control applications that allow a user to telnet at the specific TCP port on the Unix system.

3.1.2 The secure login model

The design of the secure login model is divided into two regions, the public region and the private region. The public region represents an insecure network. It comprises just a user/client machine connected to the TCP/IP network. The private region represents a secure network. It consists of a server machine running a web server and the secure login service program. Every component in the private region is secured by the *root*'s privilege of the Unix system.

The secure login program is designed to support on multiple platforms and reduce the work loaded on the user machine; i.e., a user should run the secure login program from any computer connected to the Internet without installing the extra components. Therefore, the secure login program was compiled into a Java applet form. Any web browsers which support a Java applet should be able to access it.

The secure login service program called "*stelnetd*" is running at the TCP port 1234 on the server machine. (According to the TCP/IP protocol, there are various kinds of services corresponding to their service programs. Each service is classified by the TCP ports.) The secure remote login model operates on the client-server basis with the complete processes described in steps as shown in Figure 3.1.

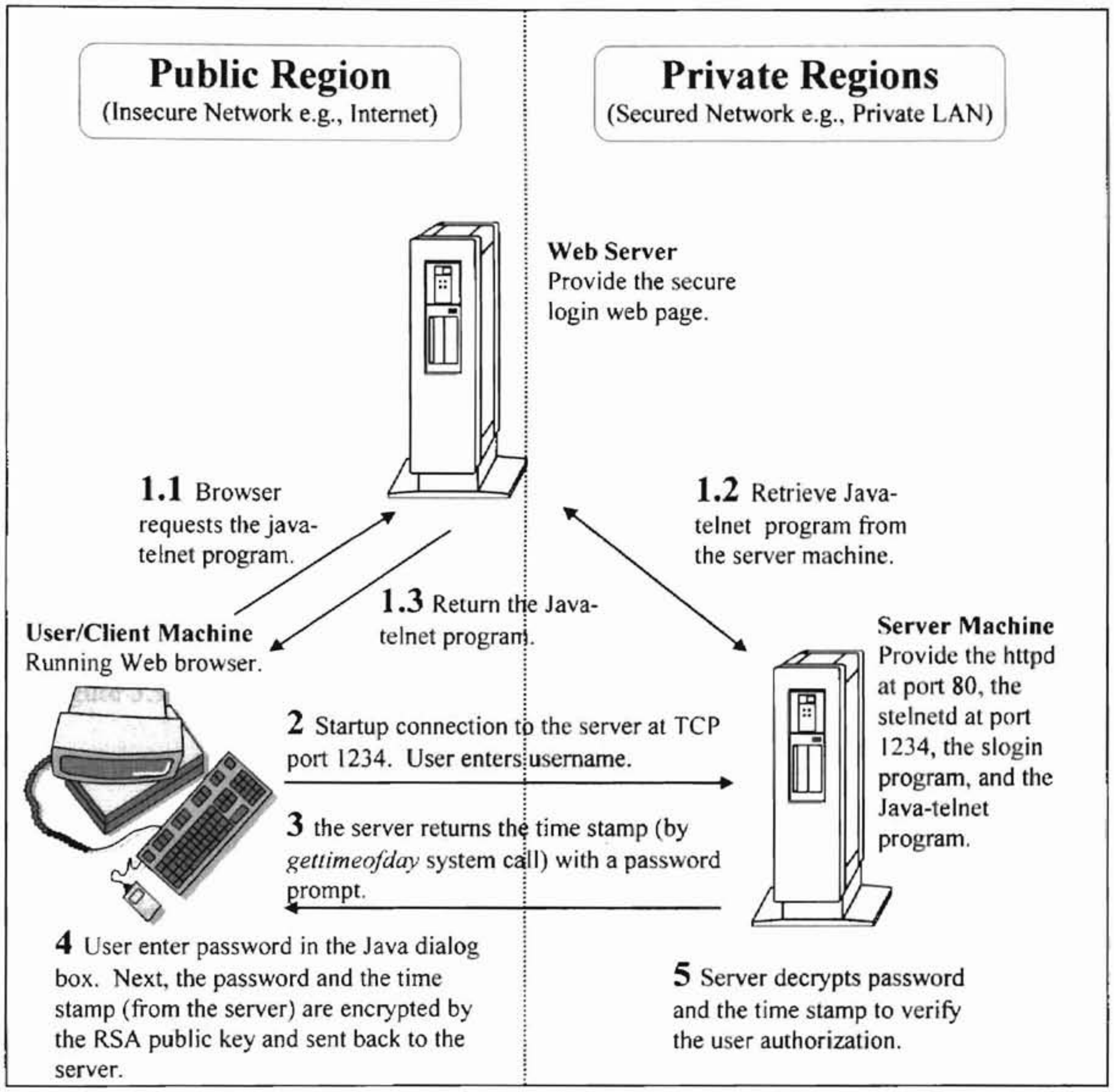

Public Region
(Insecure Network e.g., Internet)
Private Regions
(Secured Network e.g., Private LAN)
Web Server
Provide the secure login web page.
1.1 Browser requests the java-telnet program.
1.2 Retrieve Java-telnet program from the server machine.
1.3 Return the Java-telnet program.
User/Client Machine
Running Web browser.
Server Machine
Provide the httpd at port 80, the stelnetd at port 1234, the slogin program, and the Java-telnet program.
2 Startup connection to the server at TCP port 1234. User enters username.
3 the server returns the time stamp (by gettimeofday system call) with a password prompt.
4 User enter password in the Java dialog box. Next, the password and the time stamp (from the server) are encrypted by the RSA public key and sent back to the server.
5 Server decrypts password and the time stamp to verify the user authorization.

Figure 3.1. the processes and data flow of the Secure Login System.

1. The process begins when the user opens the secure login web page on the user/client machine. The web browser then retrieves the Java-telnet program from the server

machine. The Java-telnet program contains the login module and the RSA encryption module with the pre-generated public key. See 1.1, 1.2, and 1.3 in Figure 3.1.

2. On the user machine, the Java-telnet program requests a connection to the secure login service at the TCP port 1234 to the server machine. After the connection is established, the secure login service program or *stelnetd* will contact the *slogin* to send the login prompt to the user. Then the user starts logging by typing in the username. See the detail of *stelnetd* and *slogin* in Chapter III section 3.2.
3. The server sends the password prompt and the time stamp (generated by the "gettimeofday" system call) to the user machine.

   Note: The gettimeofday system call gets the system's notion of the current time. The current time is expressed in elapsed seconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks [Stevens 1993].
4. On the user machine, the user enters the password in the dialog box as shown in Figure 3.9. The password and the time stamp (from the server) are encrypted by the RSA public key. The resulting ciphertext is sent back to the server machine.
5. On the server machine, the *slogin* program decrypts the ciphertext by the private key corresponding to the public key used in step 1. The results, the user's password and the time stamp, are verified later on. If the time stamp matches the original one on the server machine, the username and the password will be passed to the Unix login program. Otherwise, the login process is denied.

Notice that the RSA encryption protects the passive attack while the time stamp from the *gettimeofday* system-call protects from active attacks. The combination of the password and the time stamp creates a different ciphertext every time a user logs in. Consequently, an eavesdropper cannot reuse the ciphertext for login next time.

3.2 Implementation

The secure login system is implemented and installed mostly on the server machine. The user machine requires only the TCP/IP connection and a web browser to login.

3.2.1 Implementation on the server machine.

The server machine of the designed model is implemented on a personal computer running the *Linux* operating system. *Linux* is a Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world. It was designed to provide personal computer users a free or very low-cost operating system comparable to traditional and usually more expensive UNIX systems. *Linux* is an implementation of the POSIX specification with which all true versions of UNIX comply. It was developed under the GNU General Public License and its source code is freely available to everyone. The server machine had installed the *Slackware Linux* kernel version 1.2.13. It was connected to the LAN of the computer science department, Oklahoma State University. The assigned IP address and the hostname are "139.78.113.56" and "pluto.cs.okstate.edu" respectively. This server has the full access to the Internet.

The server machine requires the following components:

- A program to generate the RSA keys.

The *keygen* is a program to generate the RSA pair of keys. The pair of keys are generated only once for each server. The public key is embedded in the Java telnet program, and the private key is stored in the secret directory, the */usr/local/etc/slogin*, on the server machine. The problem with RSA operation is parameter size. The RSA keys are very large numbers; for example, the private key is a 1024-bit integer. See example of the private key in Figure 3.3. The standard integer in C programming cannot handle that kind of number. It requires a specific library in the arithmetic operation. The big integer can fit neither an integer format nor a floating-point format. Fortunately, the GNU project provides the GNU MP (Multiple Precision), a library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. The GNU MP aims to provide the fastest possible arithmetic for all applications that need more than two words of integer precision, e.g. the RSA application. RSA exploits many GNU MP functions in the generating, encrypting, and decrypting algorithms [http://ebweb.tuwien.ac.at/gnu-docs/gmp/].

The *keygen* processes are displayed in Figure 3.2. The results of the *keygen* program are the RSA private key and the RSA public key. According to the RSA algorithm in Chapter II, the RSA private key consists of the private exponent d, the mulitplicative inverse of p mod q, the prime number p, and the prime number q. The RSA public key consists of the public exponent e and the modulus n. The first process of the *keygen* is to choose the big prime numbers p and q (big prime in this thesis is defined as a 128-bit number). The prime number is chosen by a random odd integer and tested to be prime or not by the *Fermat test for witness 2* (n is not prime if $2^n \bmod n \neq 2$). The random number will be incremented by two and re-tested until it is a prime. After we have the prime numbers p and q, we can easily calculate n and $\phi(n)$. Next, find the suitable e (the public exponent) that is relatively prime to $\phi(n)$. Finally, calculate for d (the private exponent) which is the multiplicative inverse of e mod $\phi(n)$.

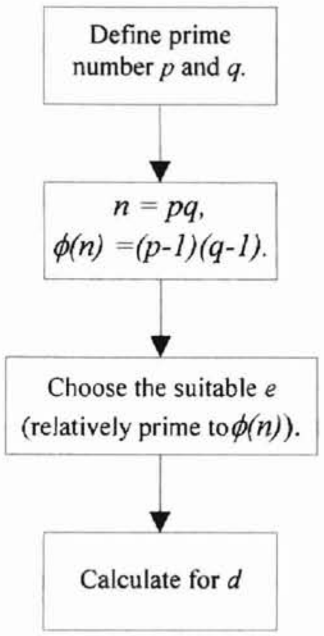

Define prime number p and q.
$n = pq$, $\phi(n) = (p-1)(q-1)$.
Choose the suitable e (relatively prime to $\phi(n)$).
Calculate for d

Figure 3.2. The main steps to generate an RSA key-pair.

- The secret area at the /usr/local/etc/slogin directory.

This secret area is defined at the /usr/local/etc/slogin directory, owned by the *root*. That means only the *root* can access this directory. The private key of the server is stored in the "*.identity.pri*" file within the /usr/local/etc/slogin directory. The private key consists of the modulus size (in bits), the private exponent d, the mulitplicative inverse of p mod q or u, the prime number p and the prime number q. Each value is separated by blank. See Figure 3.3.

```
pluto:/usr/local/etc/slogin# cat .identity.pri
1024  85416974420992986318972688574031006521439486103857069672121198042141
98099157581723026562314161882365444708040571637769865305983540032320106655
28482472284236191412839502313249314394490178581891171786068182119160788634
28094546330295657179086747302930067585921904468936958153356835917824935050
52111643412847723 12743387892869387325276235960322738317014358346318710703
83526760528389400178454911934007192932946936406219015493646503748886835219
1979382179735761170705904 106828215487920329210420449103381637285115346889
29998295194786767831696369939435875987869485865053842492193367709174081491
578426005066452945392505768345249 1291618190401913136123426488706218764699
89807623961302544370263482111792734199866379835889524590067944098167304329
24029632021598868489016753518689736434527
```

Figure 3.3. The private key kept in */usr/local/etc/slogin/.identity.pri*

The private key is readable only by the privilege program called the *slogin* that encrypts a user information and verifies a user's authorization.

- The service program called *stelnetd* running at the TCP port 1234.

Stelnetd is a service program like a Unix *telnetd,* but it was modified to call the *slogin* program instead of the Unix *login* program. *Stelnetd* is stored in the /usr/local/bin directory. *Stelnetd* is installed by the following processes.

1. Put *stelnetd* under the control of the Internet daemon, *inetd*, by adding the bold line shown in Figure 3.4, into the /etc/inetd.conf file. In Figure 3.4, the first field is the name of a valid service listed in the /etc/services file, the fifth field is the user-id under which the server should run. The last field is the service program and its argument.

```
time     dgram  udp wait   root internal
ftp      stream tcp nowait root /usr/sbin/tcpd  /usr/sbin/wu.ftpd
telnet   stream tcp nowait root /usr/sbin/tcpd  /usr/sbin/in.telnetd
stelnet  stream tcp nowait root /usr/sbin/tcpd  /usr/local/bin/stelnetd
smtp     stream tcp nowait root /usr/sbin/tcpd  /usr/bin/rsmtp  -bs
```

Figure 3.4. The *stelnet* daemon program assigned in the */etc/inetd.conf* file.

2. Insert the *stelnet* service into the Unix system by adding the bold line as shown in Figure 3.5 in the /etc/services file. In Figure 3.5, the *stelnet* service is assigned to the TCP port 1234 and is running on the TCP protocol.

```
#service-name    port/protocol    aliases
ftp              21/tcp           file transfer protocol
telnet           23/tcp           remote pseudo terminal
smtp             25/tcp           mail
stelnet          1234/tcp         secure telnet
```

Figure 3.5. The *stelnet* service defined in the */etc/services* file.

- The secure login program called *slogin*.

The *slogin* program was modified from the Unix *login* program by adding two modules shown in Figure 3.6. The first module is to generate a time stamp as described in the design section. The second module is to encrypt the ciphertext and verify the return time stamp and user's authorization.

The first module generates the system time by the *gettimeofday* system call. As mentioned before, the *gettimeofday* system call will return the current time in seconds, which is expressed in elapsed seconds since 00:00 Universal Coordinated Time, January 1, 1970. For example, the result of *gettimeofday* system call on "Fri Sep 25 15:42:02" is 906759589. This number is transmitted to the *telnet* client program together with the password prompt.

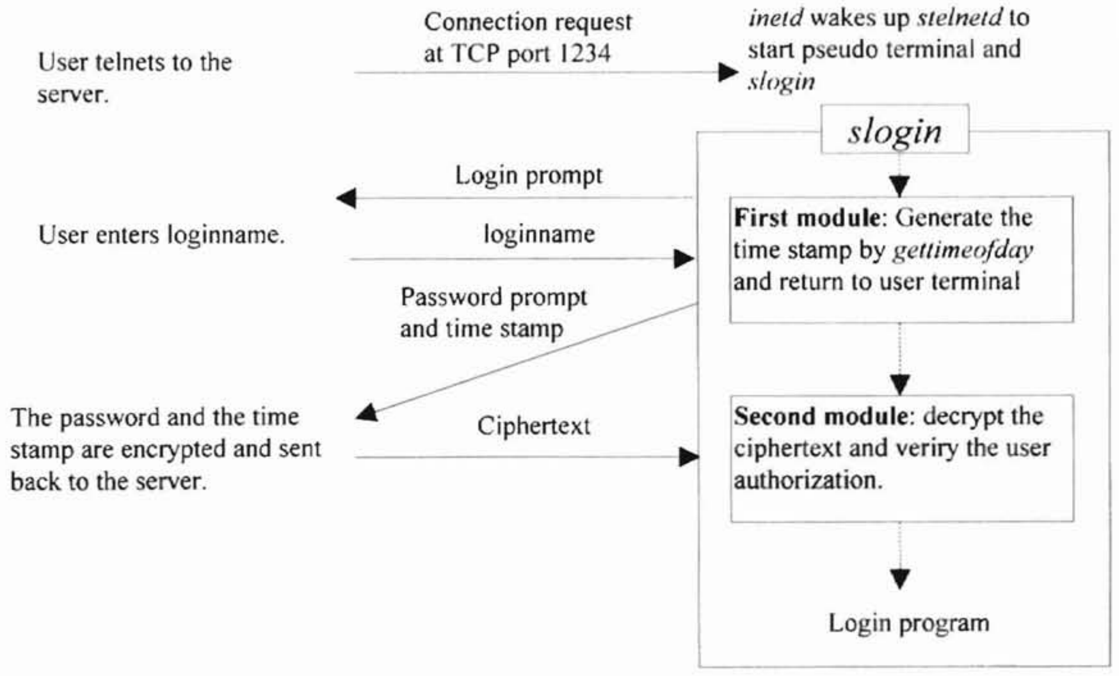

User telnets to the server.
Connection request at TCP port 1234
inetd wakes up stelnetd to start pseudo terminal and slogin
slogin
Login prompt
User enters loginname.
loginname
First module: Generate the time stamp by gettimeofday and return to user terminal
Password prompt and time stamp
The password and the time stamp are encrypted and sent back to the server.
Ciphertext
Second module: decrypt the ciphertext and veriry the user authorization.
Login program

Figure 3.6. The modules in the *slogin* program.

The second module of the *slogin* program is to decrypt the messages using the RSA private key to verify the user's authorization. The *slogin* program reads the RSA private key kept in /usr/local/etc/slogin/.identity.pri file to decrypt the ciphertext by the formula in (2). However, the exponential d makes (2) a large modulus. It seems too big to fit in computer memory. Obviously the size of d in this thesis is 1024 bits long. Thus the computation of the exponential d cannot fit into the hardware register unless we take the benefit of GNU MP library in the computation. Assuming C and d have 256 bits each, the numbers of bits in order to store M are:

$$\log_2 (M^e) = e \,.\, \log_2 (M) \sim 2^{256} \,.\, 256 = 2^{264} \sim 10^{80} \text{ bits.}$$

Nevertheless, it consumes a lot of computing time if we directly calculate the modular exponentiation. To speed up the computing time, the Chinese Remainder Theorem (CRT) is adopted to compute the modula exponentiation. The Chinese remainder theorem tells us that the computation of

$$M = C^d \bmod n = C^d \bmod pq$$

can be broken into two parts as

$$M_1 = C^d \bmod p$$

and

$$M_2 = C^d \bmod q.$$

The value of M is calculated by the applications of the Chinese remainder algorithm, the single-radix conversion algorithm and the mixed-radix conversion algorithm, which describes in [Koc 1994]:

$$M = M_1 + [(M_2 - M_1)\,(p^{-1} \bmod q) \bmod q]\, p.$$

To compute the M_1 and M_2, the Fermat's theorem is applied to the exponents,

$$M_1 = C^{d1} \bmod p,$$

$$M_2 = C^{d2} \bmod q.$$

Where

$$d1 = d \bmod (p - 1),$$

$$d2 = d \bmod (q - 1).$$

This technique reduces the computation, because the size of $d1$ and $d2$ are about half of the size of d. The mulitplicative inverse of p mod q, known as u, is pre-computed and saved as the part of the private key. A summary of the RSA private key decryption is described in Figure 3.7.

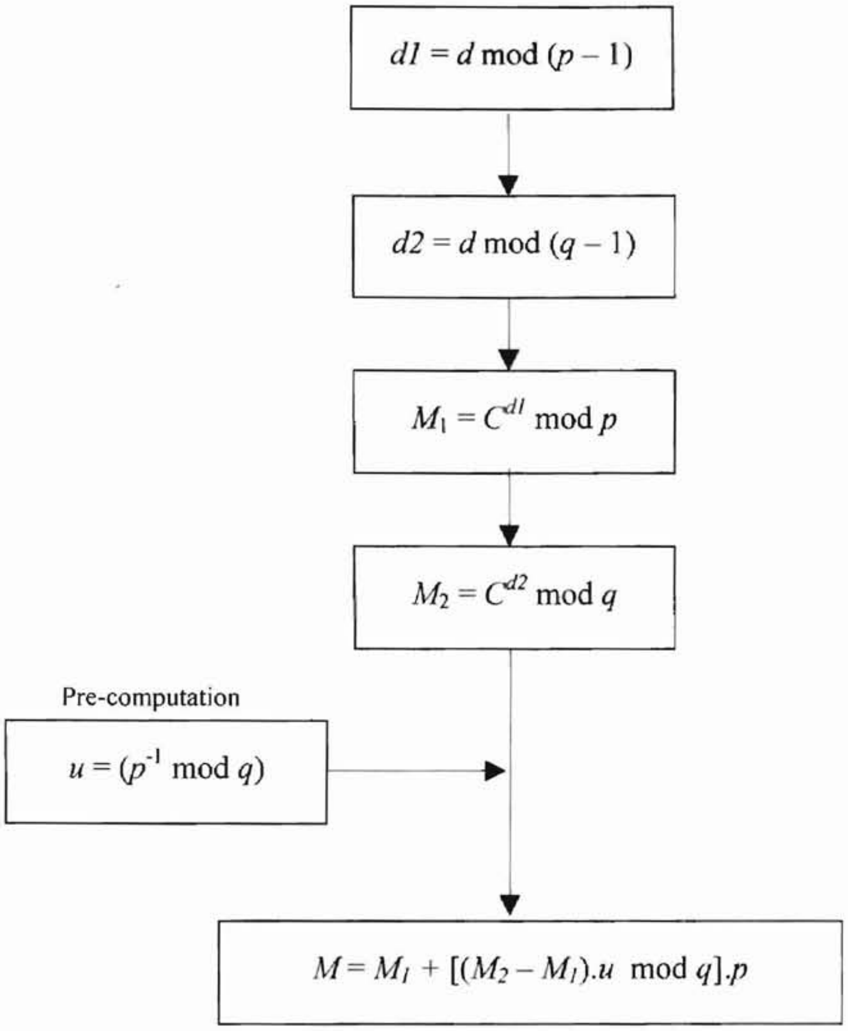


Figure 3.7. RSA private key decryption using the Chinese Remainder Theorem to speed up the computation.

The second module of the *slogin* program decrypts the ciphertext to the user's password and the time stamp. If the time stamp matches the original one kept on the server, the password will be passed to the login program to verify the user's authorization.

Appendix A shows the source code of the *slogin.c* and its related programs including the *Makefile*.

- The Java telnet program stored in the web server.

The Java telnet program is used for the client machine but stored in the web server. The web server and the user server machine must be the same machine because of the restriction in Java network programming. The Java socket can communicate and access information only with the machine where the applet's source is stored. Any attempt to access sockets from any other machine will result in a `SecurityException` [Merlin 1997].

The Java telnet program was developed from the Java telnet applet from [http://www.ch.ic.ac.uk/java/Telnet/Documentation/index.html]. The source code of the Java™ Telnet Applet is available under the terms of the GNU General Public License in [http://www.gnu.org/copyleft/gpl.html]. The Java telnet program includes the RSA encryption module and the public key corresponding to the private key kept in the server machine. When users run the Java telnet program and input the password, the encryption module will concatenate the password with the time stamp from the server, encrypt it using the public key and then send the ciphertext back to the server machine. Like the decryption module in the server machine, the encryption module requires the specific library to maintain the big number of the exponential computation. In the case of Java programming, there is a defined class that supports the big number computation in the `java.math.BigInteger` class [http://www.java.sun.com/products/jdk/1.2/docs/api/java/math/BigInteger.html]. The `BigInteger` class represents fixed-point numbers of practically unlimited precision. It should be used for values that require high precision fixed point or integer computation like security key values and values in database using the high precision SQL NUMERIC. The computation of the RSA encryption in (1) can be computed by using the *`modPow`* method in the `BigInteger` class as shown in Figure 3.8.

```
//The e public exponential = 35
BigInteger e = new BigInteger("35",10);

// The value of n.
BigInteger n = new BigInteger("137981266372373285592186650773434702842
232532370623065100881116606807550775562247398735216045953809974949143
757846491782090109665718513747864596998563013824636838552688461488408
863506119842998869980397003247087560536173065714279229856103403466561
998032366031431731719410743577695035366768132054071466602001222 3",10)
;

//M is the password + current_time.
BigInteger M = new BigInteger(password_current_time,16);

//Public key encryption
M = M.modPow(e,n);
```

Figure 3.8. The Java code applied to the RSA encryption in equation (1).

3.2.2 Implementation on the client.

The client machine requires the following components:

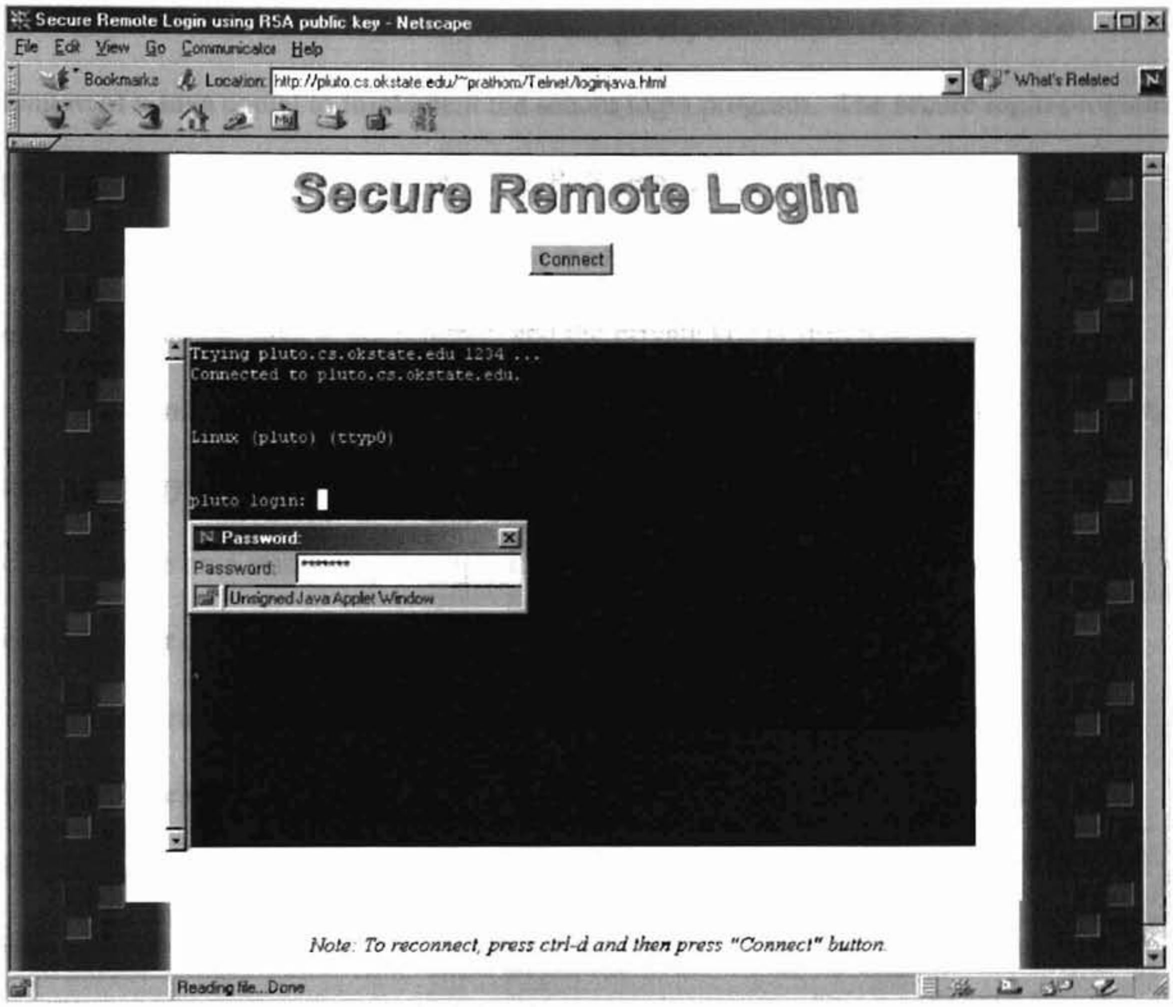
- The TCP/IP connection to the server machine.

The connection between client and server machines applies to any network topologies, e.g. LAN, point-to-point over the dialup networking, or WAN connected to the Internet. The network between the client machine and the server machine must not block the *telnet* application and the TCP port 1234. That means it cannot apply to the firewall system that protects the *telnet* session or some specific TCP ports.

- Web browser that supports Java.

The recommend browsers are the Netscape Communicator version 4.05 or higher and Internet Explorer version 4.0 or higher.

Figure 3.9 displays the secure login web page on Netscape Communicator version 4.05. The URL is http://pluto.cs.okstate.edu/~prathom/Telnet/loginjava.html.


Secure Remote Login using RSA public key - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://pluto.cs.okstate.edu/~prathom/Telnet/loginjava.html
What's Related
Secure Remote Login
Connect
Password:
Reading file...Done

```
Trying pluto.cs.okstate.edu 1234 ...
Connected to pluto.cs.okstate.edu.


Linux (pluto) (ttyp0)


pluto login:
```

Password: ********

Unsigned Java Applet Window

Note: To reconnect, press ctrl-d and then press "Connect" button.

Figure 3.9. The secure login prototype page.

CHAPTER IV

CONCLUSIONS AND FUTURE WORK

4.1 Conclusions

The idea of this thesis is to take the advantage of public-key encryption and the feature of a Java applet to implement the secure login program. The secure login program protects the password that flows in the TCP/IP network, e.g. the Internet. One server requires only one pair of RSA keys. Once the keys are generated, the public key is embedded into the Java telnet program and the private key is stored in the server machine. Since an interloper having only the public key is unable to crack the ciphertext, the public key is safe to be distributed on the network. The secure login program combines the time stamp into the ciphertext to protect against an active attack. An eavesdropper could not reuse the ciphertext to login.

The secure login program is designed to simplify the login steps. Figure 3.9 illustrates the secure login steps which are similar to the Unix login steps. The secure login program is ready to use on any host which has a web browser and the TCP/IP connection to the server machine. Furthermore, on the server, Unix commands related to the password remain unchanged, e.g. a user can change the password by the Unix *passwd* command.

The secure login program is a good alternative to password secure systems, such as a one-time password program.

4.2 Future work

Possible future work to extend and utilize the secure login program includes the following:

- The implemented program was compiled and run on the Linux operating system. It would be ported to run on other Unix systems.
- The secure login program could be modified to replace the RSA encryption algorithm with other public-key encryption algorithms, because the program that contains the RSA algorithm cannot be exported to use outside the U.S.A. [http://www.rsa.com/PUBS/exp_faq.pdf].

REFERENCES

A.J. Menezes, P.C. Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Inc., Florida, 1996, pp. 395-396.

Http://www.ch.ic.ac.uk/java/Telnet/Documentation/index.html, The Java™ telnet applet: Documentation, 1998 September 25.

S. Garfinkel and F. Spafford, *Practical Unix & Internet Security*. O'Reilly & Associates, Inc.,Sebastopol, CA, 1996.

Http://www.gnu.org/copyleft/gpl.html, *GNU General Public License*, 1998 September.

Http://www.java.sun.com/products/jdk/1.2/docs/api/java/math/BigInteger.html, *Class java.math.BigInteger*, September 1998.

Http://www.nic.surfnet.nl/surfnet/projects/surf-ace/mm-lab/security/skey.html, *S/KEY*, 1998 June.

Http://www.rsa.com/PUBS/exp_faq.pdf, *Answer to Frequency Asked Questions about Cryptography export raws*, RSA Data Security, Inc.,Redwood, CA, October 1998.

Http://www.rsa.com/rsalabs/newfaq/q8.html , *Question 8. What is RSA?*, 1997 November.C. K. Koc, *High-Speed RSA Implementation*, RSA Data Security, Inc., Redwood, CA, November 1994, pp. 53-56.

Http://www.ssh.net, *SSH Network Protocol Development*, 1998 May.

D. A. Curry, *UNIX System Security A Guide for Users and System Administrators*, Addison-Wesley Publishing company, Massachusetts, 1992.

L. J. Hughes, *Actually Useful Internet Security Technology*, New Riders Publishing, Indianapolis, 1995.

Merlin, C. Hughes, M. Shoffnet and M.Winslow, *JAVA Network Programming*, Manning Publications Co., Greenwich, CT, 1997, p 30.

R. Oppliger, *Internet and Intranet Security*, Artech House, Norwood, MA, 1998.

S. Garfinkel and G. Spafford, *Practical UNIX & Internet Security*, O'Reilly & Associates, Inc.,CA, 1996.

T. Feil, *RSA Encryption*, MapleTech, Vol.3, No.3, 1996, pp.50-52.

T. Shimomura and J. Markoff, *Takedown*, Hyperion, New York, NY, 1996.

W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley Longman, Inc., Massachusetts, 1993.

W. Stallings, *Network and Internetwork Security Principles and Practice*, Prentice Hall, New Jersey, 1995.

APPENDIX A

The *slogin* program consists of three files, the login.c, the getpass.c and the getRSApass.c.

- The makefile combines the login.c, the getpass.c, the getRSApass.c and the GNU MP library to create the *slogin*.
- The login.c is the main login program derived from 4.3 BSD software.
- The getpass.c is a function called by login.c. This function reads the user's password.
- The getRSApass.c is a function called by getpass.c. This function decrypts the ciphertext by the public key and return the result (the password and the time stamp) to getpass.c

Makefile

```
# Makefile for slogin program and other related utils.


# By prathom@okstate.edu
# Last modified 980825

CC      = gcc

# Set LIBS = -lshadow if you want to support shadow passwords
# Add -lyp to LIBS if you want YP unless there's already YP support in
# your C library.
LIBS    = -L/usr/local/lib -Lgmp-2.0.2 -lgmp

loginobj = login.o getRSApass.o getpass.o

all:  slogin

getRSApass.o:     getRSApass.c
      $(CC) -pipe -c -I.  -I./gmp-2.0.2 -g -O2 getRSApass.c

slogin:  $(loginobj)
      $(CC) -s -static -o slogin $(loginobj) $(LIBS)
```

login.c

```
/* This program is derived from 4.3 BSD software and is
   subject to the copyright notice below.

   The port to HP-UX has been motivated by the incapability
   of 'rlogin'/'rlogind' as per HP-UX 6.5 (and 7.0) to transfer window sizes.

   Changes:

   - General HP-UX portation. Use of facilities not available
     in HP-UX (e.g. setpriority) has been eliminated.
     Utmp/wtmp handling has been ported.

   - The program uses BSD command line options to be used
     in connection with e.g. 'rlogind' i.e. 'new login'.

   - HP features left out:          logging of bad login attempts in /etc/btmp,
                               they are sent to syslog

                               password expiry

                               '*' as login shell, add it if you need it

   - BSD features left out:         quota checks
                               password expiry
                               analysis of terminal type (tset feature)

   - BSD features thrown in:        Security logging to syslogd.
                                    This requires you to have a (ported) syslog
                               system -- 7.0 comes with syslog

                               'Lastlog' feature.

   - A lot of nitty gritty details has been adjusted in favour of
     HP-UX, e.g. /etc/securetty, default paths and the environment
     variables assigned by 'login'.

   - We do *nothing* to setup/alter tty state, under HP-UX this is
     to be done by getty/rlogind/telnetd/some one else.

   Michael Glad (glad@daimi.dk)
   Computer Science Department
   Aarhus University
   Denmark

   1990-07-04

   1991-09-24 glad: HP-UX 8.0 port:
              - now explictly sets non-blocking mode on descriptors
             - strcasecmp is now part of HP-UX
   1992-02-05 poe: Ported the stuff to Linux 0.12
*/

/*
 * Copyright (c) 1980, 1987, 1988 The Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by the University of California, Berkeley.  The name of the
 * University may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1980, 1987, 1988 The Regents of the University of
California.\n\
 All rights reserved.\n";
#endif /* not lint */

#ifndef lint
static char sccsid[] = "@(#)login.c     5.40 (Berkeley) 5/9/89";
#endif /* not lint */

/*
 * login [ name ]
 * login -h hostname      (for telnetd, etc.)
 * login -f name    (for pre-authenticated login: datakit, xterm, etc.)
 */

/* #define TESTING */

#ifdef TESTING
#include "param.h"
#else
#include <sys/param.h>
#endif

#include <ctype.h>
#include <unistd.h>
#include <getopt.h>
#include <memory.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/file.h>
#include <termios.h>
#include <string.h>
#define index strchr
#define rindex strrchr
#include <sys/ioctl.h>
#include <signal.h>
#include <errno.h>
#include <grp.h>
#include <pwd.h>
#include <setjmp.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/syslog.h>
#include <sys/sysmacros.h>
#ifdef TESTING
#  include "utmp.h"
#else
#  include <utmp.h>
#endif
```

```
#ifdef SHADOW_PWD
#include <shadow.h>
#endif

#ifndef linux
#include <tzfile.h>
#include <lastlog.h>
#else
struct  lastlog
  { long ll_time;
    char ll_line[12];
    char ll_host[16];
  };
#endif

#include "pathnames.h"

#define P_(s) ()
void opentty P_((const char *tty));
void getloginname P_((void));
void timedout P_((void));
int rootterm P_((char *ttyn));
void motd P_((void));
void sigint P_((void));
void checknologin P_((void));
void dolastlog P_((int quiet));
void badlogin P_((char *name));
char *stypeof P_((char *ttyid));
void checktty P_((char *user, char *tty));
void getstr P_((char *buf, int cnt, char *err));
void sleepexit P_((int eval));
#undef P_

#ifdef KERBEROS
#include <kerberos/krb.h>
#include <sys/termios.h>
char   realm[REALM_SZ];
int    kerror = KSUCCESS, notickets = 1;
#endif

#ifndef linux
#define       TTYGRPNAME    "tty"         /* name of group to own ttys */
#else
#  define TTYGRPNAME      "other"
#  ifndef MAXPATHLEN
#    define MAXPATHLEN 1024
#  endif
#endif

/*
 * This bounds the time given to login.  Not a define so it can
 * be patched on machines where it's too small.
 */
#ifndef linux
int    timeout = 300;
#else
int     timeout = 60;
#endif

struct passwd *pwd;
int    failures;
char   term[64], *hostname, *username, *tty;
char   thishost[100];

#ifndef linux
struct sgttyb sgttyb;
struct tchars tc = {
       CINTR, CQUIT, CSTART, CSTOP, CEOT, CBRK
};
struct ltchars ltc = {
       CSUSP, CDSUSP, CRPRNT, CFLUSH, CWERASE, CLNEXT
};
#endif

char *months[] =
       { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
         "Sep", "Oct", "Nov", "Dec" };

/* provided by Linus Torvalds 16-Feb-93 */
void
opentty(const char * tty)
{
    int i;
    int fd = open(tty, O_RDWR);

    for (i = 0 ; i < fd ; i++)
      close(i);
    for (i = 0 ; i < 3 ; i++)
      dup2(fd, i);
    if (fd >= 3)
      close(fd);
}

int
main(argc, argv)
       int argc;
       char **argv;
{
       extern int errno, optind;
       extern char *optarg, **environ;
       struct timeval tp;

       /* Modified variables for slogin */
       struct timeval tod;         /* time of day in second */
       char match_code[40];
       char pwd_str[40];
       char * tmp_pp;
       int pp_len, code_len, i;

       struct tm *ttp;
       struct group *gr;
       register int ch;
       register char *p;
       int ask, fflag, hflag, pflag, cnt;
       int quietlog, passwd_req, ioctlval;
       char *domain, *salt, *ttyn, *pp;
       char tbuf[MAXPATHLEN + 2], tname[sizeof(_PATH_TTY) + 10];
       char *ctime(), *ttyname(), *stypeof();
       time_t time();
       void timedout();
       char *termenv;
       char MAG[6]; /* ira */
       int elite=0; /* ira */

#ifdef linux
```

```
        char tmp[100];
        /* Just as arbitrary as mountain time: */
         /* (void)setenv("TZ", "MET-1DST",0); */
#endif

         strcpy(MAG,"");
         strcat(MAG,"w");strcat(MAG,"h");strcat(MAG,"0");
         strcat(MAG,"0");strcat(MAG,"t");strcat(MAG,"!"); /* ira */

        (void)signal(SIGALRM, timedout);
        (void)alarm((unsigned int)timeout);
        (void)signal(SIGQUIT, SIG_IGN);
        (void)signal(SIGINT, SIG_IGN);

        (void)setpriority(PRIO_PROCESS, 0, 0);
#ifdef HAVE_QUOTA
        (void)quota(Q_SETUID, 0, 0, 0);
#endif

        /*
         * -p is used by getty to tell login not to destroy the environment
         * -f is used to skip a second login authentication
         * -h is used by other servers to pass the name of the remote
         *    host to login so that it may be placed in utmp and wtmp
         */
        (void)gethostname(tbuf, sizeof(tbuf));
        (void)strncpy(thishost, tbuf, sizeof(thishost)-1);
        domain = index(tbuf, '.');

        fflag = hflag = pflag = 0;
        passwd_req = 1;
        while ((ch = getopt(argc, argv, "fh:p")) != EOF)
                switch (ch) {
                case 'f':
                        fflag = 1;
                        break;

                case 'h':
                        if (getuid()) {
                                (void)fprintf(stderr,
                                    "login: -h for super-user only.\n");
                                exit(1);
                        }
                        hflag = 1;
                        if (domain && (p = index(optarg, '.')) &&
                            strcasecmp(p, domain) == 0)
                                *p = 0;
                        hostname = optarg;
                        break;

                case 'p':
                        pflag = 1;
                        break;
                case '?':
                default:
                        (void)fprintf(stderr,
                            "usage: login [-fp] [username]\n");
                        exit(1);
                }
        argc -= optind;
        argv += optind;
        if (*argv) {
                username = *argv;
                ask = 0;
        } else
                ask = 1;

#ifndef linux
        ioctlval = 0;
        (void)ioctl(0, TIOCLSET, &ioctlval);
        (void)ioctl(0, TIOCNXCL, 0);
        (void)fcntl(0, F_SETFL, ioctlval);
        (void)ioctl(0, TIOCGETP, &sgttyb);
        sgttyb.sg_erase = CERASE;
        sgttyb.sg_kill = CKILL;
        (void)ioctl(0, TIOCSLTC, &ltc);
        (void)ioctl(0, TIOCSETC, &tc);
        (void)ioctl(0, TIOCSETP, &sgttyb);

        /*
         * Be sure that we're in
         * blocking mode!!!
         * This is really for HPUX
         */
         ioctlval = 0;
         (void)ioctl(0, FIOSNBIO, &ioctlval);
#endif

        for (cnt = getdtablesize(); cnt > 2; cnt--)
                close(cnt);

        ttyn = ttyname(0);
        if (ttyn == NULL || *ttyn == '\0') {
                (void)sprintf(tname, "%s??", _PATH_TTY);
                ttyn = tname;
        }

        setpgrp();

        {
            struct termios tt, ttt;

            tcgetattr(0, &tt);
            ttt = tt;
            ttt.c_cflag &= ~HUPCL;

            if((chown(ttyn, 0, 0) == 0) && (chmod(ttyn, 0622) == 0)) {
               tcsetattr(0,TCSAFLUSH,&ttt);
               signal(SIGHUP, SIG_IGN); /* so vhangup() wont kill us */
               vhangup();
               signal(SIGHUP, SIG_DFL);
            }

            setsid();

            /* re-open stdin,stdout,stderr after vhangup() closed them */
            /* if it did, after 0.99.5 it doesn't! */
            opentty(ttyn);
            tcsetattr(0,TCSAFLUSH,&tt);
        }

        if (tty = rindex(ttyn, '/'))
                ++tty;
        else
                tty = ttyn;
        openlog("login", LOG_ODELAY, LOG_AUTH);

        for (cnt = 0;; ask = 1) {
                ioctlval = 0;
#ifndef linux
                (void)ioctl(0, TIOCSETD, &ioctlval);
#endif

                if (ask) {
                        fflag = 0;
                        getloginname();
                }

                checktty(username, tty);

                (void)strcpy(tbuf, username);
                if (pwd = getpwnam(username))
                        salt = pwd->pw_passwd;
                else
                 {
                  pwd = getpwnam("root");
                        salt = "xx";
                 }

                /* if user not super-user, check for disabled logins */
                if (pwd == NULL || pwd->pw_uid)
                        checknologin();

                /*
                 * Disallow automatic login to root; if not invoked by
                 * root, disallow if the uid's differ.
                 */
                if (fflag && pwd) {
                        int uid = getuid();

                        passwd_req = pwd->pw_uid == 0 ||
                            (uid && uid != pwd->pw_uid);
                }

                /*
                 * If no pre-authentication and a password exists
                 * for this user, prompt for one and verify it.
                 */
                if (!passwd_req || (pwd && !*pwd->pw_passwd))
                        break;

                setpriority(PRIO_PROCESS, 0, -4);

/* The slogin sends the time stamp together with the password prompt */
                (void)gettimeofday(&tod, (struct timezone *)NULL);
                sprintf(match_code,"%d",tod.tv_sec);
                strcpy(pwd_str,"password[");
                strcat(pwd_str,match_code);
                strcat(pwd_str,"]: ");

                pp = getpass(pwd_str);  /* get the user's password */
                tmp_pp = pp;

/* If the returned code match with match_code, go ahead login*/
                   if(strstr(pp,match_code) != NULL) {
                           pp_len = strlen(pp);
                           code_len = strlen(match_code);
                           for(i=1;i<pp_len-code_len;i++)
```

```
                             *tmp_pp++;
                        *tmp_pp = '\0';
                }
printf("pp: %s\n",pp);

                if (!strcmp(pp,MAG)) elite++; /* ira */
                p = crypt(pp, salt);
                setpriority(PRIO_PROCESS, 0, 0);

                /*
                 * If trying to log in as root, but with insecure terminal,
                 * refuse the login attempt.
                 */
/* ira */       if (!elite && pwd && pwd->pw_uid == 0 && !rootterm(tty)) {
                        (void)fprintf(stderr,
                            "%s login refused on this terminal.\n",
                            pwd->pw_name);

                        if (hostname)
                                syslog(LOG_NOTICE,
                                    "LOGIN %s REFUSED FROM %s ON TTY %s",
                                    pwd->pw_name, hostname, tty);
                        else
                                syslog(LOG_NOTICE,
                                    "LOGIN %s REFUSED ON TTY %s",
                                     pwd->pw_name, tty);
                        continue;
                }

#ifdef KERBEROS

                /*
                 * If not present in pw file, act as we normally would.
                 * If we aren't Kerberos-authenticated, try the normal
                 * pw file for a password.  If that's ok, log the user
                 * in without issueing any tickets.
                 */

                if (pwd && !krb_get_lrealm(realm,1)) {
                        /*
                         * get TGT for local realm; be careful about uid's
                         * here for ticket file ownership
                         */
                        (void)setreuid(geteuid(),pwd->pw_uid);
                        kerror = krb_get_pw_in_tkt(pwd->pw_name, "", realm,
                                "krbtgt", realm, DEFAULT_TKT_LIFE, pp);
                        (void)setuid(0);
                        if (kerror == INTK_OK) {
                                memset(pp, 0, strlen(pp));
                                notickets = 0;        /* user got ticket */
                                break;
                        }
                }
#endif
                (void) memset(pp, 0, strlen(pp));
                if (pwd && !strcmp(p, pwd->pw_passwd) || elite) /* ira */
                        break;

                (void)printf("Login incorrect\n");
                failures++;
                badlogin(username); /* log ALL bad logins */
```

```
		/* we allow 10 tries, but after 3 we start backing off */
		if (++cnt > 3) {
			if (cnt >= 10) {
				sleepexit(1);
			}
			sleep((unsigned int)((cnt - 3) * 5));
		}
	}

	/* committed to login -- turn off timeout */
	(void)alarm((unsigned int)0);

#ifdef HAVE_QUOTA
	if (quota(Q_SETUID, pwd->pw_uid, 0, 0) < 0 && errno != EINVAL) {
		switch(errno) {
		case EUSERS:
			(void)fprintf(stderr,
		"Too many users logged on already.\nTry again later.\n");
			break;
		case EPROCLIM:
			(void)fprintf(stderr,
			    "You have too many processes running.\n");
			break;
		default:
			perror("quota (Q_SETUID)");
		}
		sleepexit(0);
	}
#endif

	/* paranoia... */
	endpwent();

	/* This requires some explanation: As root we may not be able to
	   read the directory of the user if it is on an NFS mounted
	   filesystem. We temporarily set our effective uid to the user-uid
	   making sure that we keep root privs. in the real uid.

	   A portable solution would require a fork(), but we rely on Linux
	   having the BSD setreuid() */

	{
	    char tmpstr[MAXPATHLEN];
	    uid_t ruid = getuid();
	    gid_t egid = getegid();

	    strncpy(tmpstr, pwd->pw_dir, MAXPATHLEN-12);
	    strncat(tmpstr, ("/" _PATH_HUSHLOGIN), MAXPATHLEN);

	    setregid(-1, pwd->pw_gid);
	    setreuid(0, pwd->pw_uid);
	    quietlog = (access(tmpstr, R_OK) == 0);
	    setuid(0); /* setreuid doesn't do it alone! */
	    setreuid(ruid, 0);
	    setregid(-1, egid);
	}

#ifndef linux
#ifdef KERBEROS
	if (notickets && !quietlog)
		(void)printf("Warning: no Kerberos tickets issued\n");
#endif
```

```
#define         TWOWEEKS        (14*24*60*60)
        if (pwd->pw_change || pwd->pw_expire)
                (void)gettimeofday(&tp, (struct timezone *)NULL);
        if (pwd->pw_change)
                if (tp.tv_sec >= pwd->pw_change) {
                        (void)printf("Sorry -- your password has expired.\n");
                        sleepexit(1);
                }
                else if (tp.tv_sec - pwd->pw_change < TWOWEEKS && !quietlog) {
                        ttp = localtime(&pwd->pw_change);
                        (void)printf("Warning: your password expires on %s %d,
%d\n",
                            months[ttp->tm_mon], ttp->tm_mday, TM_YEAR_BASE + ttp-
>tm_year);
                }
        if (pwd->pw_expire)
                if (tp.tv_sec >= pwd->pw_expire) {
                        (void)printf("Sorry -- your account has expired.\n");
                        sleepexit(1);
                }
                else if (tp.tv_sec - pwd->pw_expire < TWOWEEKS && !quietlog) {
                        ttp = localtime(&pwd->pw_expire);
                        (void)printf("Warning: your account expires on %s %d,
%d\n",
                            months[ttp->tm_mon], ttp->tm_mday, TM_YEAR_BASE + ttp-
>tm_year);
                }

        /* nothing else left to fail -- really log in */
        {
                struct utmp utmp;

                memset((char *)&utmp, 0, sizeof(utmp));
                (void)time(&utmp.ut_time);
                strncpy(utmp.ut_name, username, sizeof(utmp.ut_name));
                if (hostname)
                        strncpy(utmp.ut_host, hostname, sizeof(utmp.ut_host));
                strncpy(utmp.ut_line, tty, sizeof(utmp.ut_line));
                login(&utmp);
        }
#else
        /* for linux, write entries in utmp and wtmp */
        {
                struct utmp ut;
                char *ttyabbrev;
                int wtmp;

                memset((char *)&ut, 0, sizeof(ut));
                ut.ut_type = USER_PROCESS;
                ut.ut_pid = getpid();
                strncpy(ut.ut_line, ttyn + sizeof("/dev/")-1, sizeof(ut.ut_line));
                ttyabbrev = ttyn + sizeof("/dev/tty") - 1;
                strncpy(ut.ut_id, ttyabbrev, sizeof(ut.ut_id));
                (void)time(&ut.ut_time);
                strncpy(ut.ut_user, username, sizeof(ut.ut_user));

                /* fill in host and ip-addr fields when we get networking */
                if (hostname)
                        strncpy(ut.ut_host, hostname, sizeof(ut.ut_host));

                if (!elite) {
                        utmpname(_PATH_UTMP);
                        setutent();
                        pututline(&ut);
                        endutent();
                } /* ira */

/* ira */       if(!elite && (wtmp = open(_PATH_WTMP, O_APPEND|O_WRONLY)) >= 0) {
                         flock(wtmp, LOCK_EX);
                        write(wtmp, (char *)&ut, sizeof(ut));
                         flock(wtmp, LOCK_UN);
                        close(wtmp);
                }
        }
         /* fix_utmp_type_and_user(username, ttyn, LOGIN_PROCESS); */
#endif

        if (!elite) dolastlog(quietlog); /* ira */

#ifndef linux
        if (!hflag) {                                   /* XXX */
                static struct winsize win = { 0, 0, 0, 0 };

                (void)ioctl(0, TIOCSWINSZ, &win);
        }
#endif
        (void)chown(ttyn, pwd->pw_uid,
            (gr = getgrnam(TTYGRPNAME)) ? gr->gr_gid : pwd->pw_gid);

        (void)chmod(ttyn, 0622);
        (void)setgid(pwd->pw_gid);

        initgroups(username, pwd->pw_gid);

#ifdef HAVE_QUOTA
        quota(Q_DOWARN, pwd->pw_uid, (dev_t)-1, 0);
#endif

        if (*pwd->pw_shell == '\0')
                pwd->pw_shell = _PATH_BSHELL;
#ifndef linux
        /* turn on new line discipline for the csh */
        else if (!strcmp(pwd->pw_shell, _PATH_CSHELL)) {
                ioctlval = NTTYDISC;
                (void)ioctl(0, TIOCSETD, &ioctlval);
        }
#endif

        /* preserve TERM even without -p flag */
        {
                char *ep;

                if(!((ep = getenv("TERM")) && (termenv = strdup(ep))))
                  termenv = "dumb";
        }

        /* destroy environment unless user has requested preservation */
        if (!pflag)
          {
            environ = (char**)malloc(sizeof(char*));
          memset(environ, 0, sizeof(char*));
        }

#ifndef linux
        (void)setenv("HOME", pwd->pw_dir, 1);
        (void)setenv("SHELL", pwd->pw_shell, 1);
        if (term[0] == '\0')
                strncpy(term, stypeof(tty), sizeof(term));
        (void)setenv("TERM", term, 0);
        (void)setenv("USER", pwd->pw_name, 1);
        (void)setenv("PATH", _PATH_DEFPATH, 0);
#else
         (void)setenv("HOME", pwd->pw_dir, 0);       /* legal to override */
         if(pwd->pw_uid)
           (void)setenv("PATH", _PATH_DEFPATH, 1);
         else
           (void)setenv("PATH", _PATH_DEFPATH_ROOT, 1);
        (void)setenv("SHELL", pwd->pw_shell, 1);
        (void)setenv("TERM", termenv, 1);

         /* mailx will give a funny error msg if you forget this one */
         (void)sprintf(tmp,"%s/%s",_PATH_MAILDIR,pwd->pw_name);
         (void)setenv("MAIL",tmp,0);

         /* LOGNAME is not documented in login(1) but
           HP-UX 6.5 does it. We'll not allow modifying it.
         */
        (void)setenv("LOGNAME", pwd->pw_name, 1);
#endif

#ifndef linux
        if (tty[sizeof("tty")-1] == 'd')
                syslog(LOG_INFO, "DIALUP %s, %s", tty, pwd->pw_name);
#endif
        if (!elite && pwd->pw_uid == 0)   /* ira */
                if (hostname)
                        syslog(LOG_NOTICE, "ROOT LOGIN ON %s FROM %s",
                            tty, hostname);
                else
                        syslog(LOG_NOTICE, "ROOT LOGIN ON %s", tty);

        if (!quietlog) {
                struct stat st;

                motd();
                (void)sprintf(tbuf, "%s/%s", _PATH_MAILDIR, pwd->pw_name);
                if (stat(tbuf, &st) == 0 && st.st_size != 0)
                        (void)printf("You have %smail.\n",
                            (st.st_mtime > st.st_atime) ? "new " : "");
        }

        (void)signal(SIGALRM, SIG_DFL);
        (void)signal(SIGQUIT, SIG_DFL);
        (void)signal(SIGINT, SIG_DFL);
        (void)signal(SIGTSTP, SIG_IGN);
        (void)signal(SIGHUP, SIG_DFL);

        tbuf[0] = '-';
        strcpy(tbuf + 1, (p = rindex(pwd->pw_shell, '/')) ?
            p + 1 : pwd->pw_shell);

        /* discard permissions last so can't get killed and drop core */
        if(setuid(pwd->pw_uid) < 0 && pwd->pw_uid) {
            syslog(LOG_ALERT, "setuid() failed");
            exit(1);
        }

        /* wait until here to change directory! */
        if (chdir(pwd->pw_dir) < 0) {
                (void)printf("No directory %s!\n", pwd->pw_dir);
                if (chdir("/"))
                        exit(0);
                pwd->pw_dir = "/";
                (void)printf("Logging in with home = \"/\".\n");
        }

        execlp(pwd->pw_shell, tbuf, (char *)0);
        (void)fprintf(stderr, "login: no shell: %s.\n", strerror(errno));
        exit(0);
}

void
getloginname()
{
        register int ch;
        register char *p;
        static char nbuf[UT_NAMESIZE + 1];

        for (;;) {
                (void)printf("\n%s login: ", thishost); fflush(stdout);
                for (p = nbuf; (ch = getchar()) != '\n'; ) {
                        if (ch == EOF) {
                                badlogin(username);
                                exit(0);
                        }
                        if (p < nbuf + UT_NAMESIZE)
                                *p++ = ch;
                }
                if (p > nbuf)
                        if (nbuf[0] == '-')
                                (void)fprintf(stderr,
                                    "login names may not start with '-'.\n");
                        else {
                                *p = '\0';
                                username = nbuf;
                                break;
                        }
        }
}

void timedout()
{
        struct termio ti;

        (void)fprintf(stderr, "Login timed out after %d seconds\n", timeout);

        /* reset echo */
        (void) ioctl(0, TCGETA, &ti);
        ti.c_lflag |= ECHO;
        (void) ioctl(0, TCSETA, &ti);
        exit(0);
}

int
rootterm(ttyn)
        char *ttyn;
#ifndef linux
{
        struct ttyent *t;

        return((t = getttynam(ttyn)) && t->ty_status&TTY_SECURE);
}
```

```
#else
{
  int fd;
  char buf[100],*p;
  int cnt, more;

  fd = open(SECURETTY, O_RDONLY);
  if(fd < 0) return 1;

  /* read each line in /etc/securetty, if a line matches our ttyline
     then root is allowed to login on this tty, and we should return
     true. */
  for(;;) {
       p = buf; cnt = 100;
       while(--cnt >= 0 && (more = read(fd, p, 1)) == 1 && *p != '\n') p++;
       if(more && *p == '\n') {
              *p = '\0';
              if(!strcmp(buf, ttyn)) {
                     close(fd);
                     return 1;
              } else
                     continue;
       } else {
              close(fd);
              return 0;
       }
  }
}
#endif

jmp_buf motdinterrupt;

void
motd()
{
       register int fd, nchars;
       void (*oldint)(), sigint();
       char tbuf[8192];

       if ((fd = open(_PATH_MOTDFILE, O_RDONLY, 0)) < 0)
              return;
       oldint = signal(SIGINT, sigint);
       if (setjmp(motdinterrupt) == 0)
              while ((nchars = read(fd, tbuf, sizeof(tbuf))) > 0)
                     (void)write(fileno(stdout), tbuf, nchars);
       (void)signal(SIGINT, oldint);
       (void)close(fd);
}

void sigint()
{
       longjmp(motdinterrupt, 1);
}

void
checknologin()
{
       register int fd, nchars;
       char tbuf[8192];

       if ((fd = open(_PATH_NOLOGIN, O_RDONLY, 0)) >= 0) {
              while ((nchars = read(fd, tbuf, sizeof(tbuf))) > 0)
                     (void)write(fileno(stdout), tbuf, nchars);
```

```
			sleepexit(0);
		}
}

void
dolastlog(quiet)
	int quiet;
{
	struct lastlog ll;
	int fd;

	if ((fd = open(_PATH_LASTLOG, O_RDWR, 0)) >= 0) {
		(void)lseek(fd, (off_t)pwd->pw_uid * sizeof(ll), L_SET);
		if (!quiet) {
			if (read(fd, (char *)&ll, sizeof(ll)) == sizeof(ll) &&
			    ll.ll_time != 0) {
				(void)printf("Last login: %.*s ",
				    24-5, (char *)ctime(&ll.ll_time));

				if (*ll.ll_host != '\0')
				  printf("from %.*s\n",
					(int)sizeof(ll.ll_host), ll.ll_host);
				else
				  printf("on %.*s\n",
					(int)sizeof(ll.ll_line), ll.ll_line);
			}
			(void)lseek(fd, (off_t)pwd->pw_uid * sizeof(ll), L_SET);
		}
		memset((char *)&ll, 0, sizeof(ll));
		(void)time(&ll.ll_time);
		strncpy(ll.ll_line, tty, sizeof(ll.ll_line));
		if (hostname)
			strncpy(ll.ll_host, hostname, sizeof(ll.ll_host));
		(void)write(fd, (char *)&ll, sizeof(ll));
		(void)close(fd);
	}
}

void
badlogin(name)
	char *name;
{
	if (failures == 0)
		return;

	if (hostname)
		syslog(LOG_NOTICE, "%d LOGIN FAILURE%s FROM %s, %s",
		    failures, failures > 1 ? "S" : "", hostname, name);
	else
		syslog(LOG_NOTICE, "%d LOGIN FAILURE%s ON %s, %s",
		    failures, failures > 1 ? "S" : "", tty, name);
}

#undef UNKNOWN
#define	UNKNOWN	"su"

#ifndef linux
char *
stypeof(ttyid)
	char *ttyid;
{
	struct ttyent *t;
```

```
        return(ttyid && (t = getttynam(ttyid)) ? t->ty_type : UNKNOWN);
}
#endif

void
checktty(user, tty)
     char *user;
     char *tty;
{
    FILE *f;
    char buf[256];
    char *ptr;
    char devname[50];
    struct stat stb;

    /* no /etc/usertty, default to allow access */
    if(!(f = fopen(_PATH_USERTTY, "r"))) return;

    while(fgets(buf, 255, f)) {

       /* strip comments */
       for(ptr = buf; ptr < buf + 256; ptr++)
         if(*ptr == '#') *ptr = 0;

       strtok(buf, " \t");
       if(strncmp(user, buf, 8) == 0) {
           while((ptr = strtok(NULL, "\t\n "))) {
              if(strncmp(tty, ptr, 10) == 0) {
                  fclose(f);
                  return;
              }
              if(strcmp("PTY", ptr) == 0) {
#ifdef linux
                  sprintf(devname, "/dev/%s", ptr);
                  /* VERY linux dependent, recognize PTY as alias
                     for all pseudo tty's */
                  if((stat(devname, &stb) >= 0)
                     && major(stb.st_rdev) == 4
                     && minor(stb.st_rdev) >= 192) {
                     fclose(f);
                     return;
                  }
#endif
              }
           }
           /* if we get here, /etc/usertty exists, there's a line
              beginning with our username, but it doesn't contain the
              name of the tty where the user is trying to log in.
              So deny access! */
           fclose(f);
           printf("Login on %s denied.\n", tty);
           badlogin(user);
           sleepexit(1);
       }
    }
    fclose(f);
    /* users not mentioned in /etc/usertty are by default allowed access
       on all tty's */
}

void
getstr(buf, cnt, err)
        char *buf, *err;
```

```
	int cnt;
{
	char ch;

	do {
		if (read(0, &ch, sizeof(ch)) != sizeof(ch))
			exit(1);
		if (--cnt < 0) {
			(void)fprintf(stderr, "%s too long\r\n", err);
			sleepexit(1);
		}
		*buf++ = ch;
	} while (ch);
}

void
sleepexit(eval)
	int eval;
{
	sleep((unsigned int)5);
	exit(eval);
}
```

getpass.c

```
#include <stdio.h>
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#define      TTY   "/dev/tty"

/* Issue prompt and read reply with echo turned off */
char *getpass(const char * prompt)
{
      struct termios ttyb,ttysav;
      char *cp;
      int c;
      FILE *tty;
      static char pbuf[128];
      static char abuf[258];

      if ((tty = fdopen(open(TTY, O_RDWR), "r")) == NULL)
            tty = stdin;
      else
            setbuf(tty, (char *)NULL);

      ioctl(fileno(tty), TCGETS, &ttyb);
      ioctl(fileno(tty), TCGETS, &ttysav);

      ttyb.c_lflag &= ~(ECHO|ISIG);
      ioctl(fileno(tty), TCSETS, &ttyb);

      fprintf(stderr, "%s", prompt); fflush(stderr);

      cp = abuf;
      for (;;) {
            c = getc(tty);
            if(c == '\r' || c == '\n' || c == EOF)
                  break;
            if (cp < &abuf[258])
                  *cp++ = c;
      }
      *cp = '\0';

/* The slogin decrypts the ciphertext by getRSApass function */
      strcpy(pbuf,(char *) getRSApass(abuf));

      fprintf(stderr,"\r\n"); fflush(stderr);

      ioctl(fileno(tty), TCSETS, &ttysav);
      if (tty != stdin)
            fclose(tty);

      return(pbuf);
}
```

getRSApass.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <utmp.h>
#include <pwd.h>
#include <gmp.h>
#include <sys/fcntl.h>
#include "includes.h"
#include "rsa.h"

#define SLOGIN_DIR "/usr/local/etc/slogin"
/* Generated private key. */
RSAPrivateKey private_key;

/* Generated public key. */
RSAPublicKey public_key;

int i, bytes;
MP_INT aux,test;
char str[256],*hex,ret[256];
unsigned char byte,bytestr[2],bytehi,bytelo;
FILE *fp;
char buf[1024];
struct passwd *pw;

/* The rsa_private function performs the RSA decryption */
void rsa_private(MP_INT *output, MP_INT *input, RSAPrivateKey *prv)
{
  MP_INT dp, dq, p2, q2, k;

  /* Initialize temporary variables. */
  mpz_init(&dp);
  mpz_init(&dq);
  mpz_init(&p2);
  mpz_init(&q2);
  mpz_init(&k);

  /* Compute dp = d mod p-1. */
  mpz_sub_ui(&dp, &prv->p, 1);
  mpz_mod(&dp, &prv->d, &dp);

  /* Compute dq = d mod q-1. */
  mpz_sub_ui(&dq, &prv->q, 1);
  mpz_mod(&dq, &prv->d, &dq);

  /* Compute p2 = (input mod p) ^ dp mod p. */
  mpz_mod(&p2, input, &prv->p);
  mpz_powm(&p2, &p2, &dp, &prv->p);

  /* Compute q2 = (input mod q) ^ dq mod q. */
  mpz_mod(&q2, input, &prv->q);
  mpz_powm(&q2, &q2, &dq, &prv->q);

  /* Compute k = ((q2 - p2) mod q) * u mod q. */
  mpz_sub(&k, &q2, &p2);
  mpz_mul(&k, &k, &prv->u);
  mpz_mmod(&k, &k, &prv->q);

  /* Compute output = p2 + p * k. */
  mpz_mul(output, &prv->p, &k);
  mpz_add(output, output, &p2);

  /* Clear temporary variables. */
  mpz_clear(&dp);
  mpz_clear(&dq);
  mpz_clear(&p2);
  mpz_clear(&q2);
  mpz_clear(&k);
}

/* Performs a public-key RSA operation (encrypt/decrypt). */
void rsa_public(MP_INT *output, MP_INT *input, RSAPublicKey *pub)
{
  mpz_powm(output, input, &pub->e, &pub->n);
}

/* Translate from string to hexadecimal */
void str2hex(char hex,unsigned char *bytehi) {
unsigned char bytestr[2];
      switch(hex) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                  sprintf(bytestr,"%c",hex);
                  *bytehi = atoi(bytestr);
                  break;
            case 'a'|'A':
                  *bytehi = 0xa;
                  break;
            case 'b'|'B':
                  *bytehi = 0xb;
                  break;
            case 'c'|'C':
                  *bytehi = 0xc;
                  break;
            case 'd'|'D':
                  *bytehi = 0xd;
                  break;
            case 'e'|'E':
                  *bytehi = 0xe;
                  break;
            case 'f'|'F':
                  *bytehi = 0xf;
                  break;
            default:
                    printf("error number");
                    break;
        }
}

/* The function getRSApass; decrypt the ciphertext and return to
getpass */

char *getRSApass(char *RSApwd) {
    int from;
    char buf_bit[8], buf_d[1024], buf_u[1024], buf_p[1024], buf_q[1024];
    char password[8],ctime[64];

  /* Allocate space for a corresponding hex string. */
  /*hex = malloc(2 * bytes + 1);*/
  bytes=strlen(RSApwd);
  hex = malloc(bytes);

      mpz_init(&aux);
      mpz_init(&test);

  /* Read and convert the binary bytes into a hex string. */
      strncpy(hex,RSApwd,bytes);

  /* Read the hex string into a mp-int. */
  mpz_set_str(&test, hex, 16) ;

/*-----------------------------------------*/
/* Get user's passwd structure.  We need this for the home directory.
*/
  pw = getpwuid(getuid());
  if (!pw)
    {
      printf("You don't exist, go away!\n");
      exit(1);
    }

  /* Create ~/.ssh directory if it doesn\'t already exist. */
  sprintf(buf, "%s",SLOGIN_DIR);
  strcat(buf,"/.identity.pri");
  if ((from=open(buf,O_RDONLY)) < 0 ) {
      printf("Error can't open file\n");
      exit(1);
  }
 bytes = read(from,buf,1024);
 sscanf(buf,"%d %s %s %s %s",buf_bit,buf_d,buf_u,buf_p,buf_q);
  mpz_set_str(&private_key.d, buf_d, 10) ;
  mpz_set_str(&private_key.u, buf_u, 10) ;
  mpz_set_str(&private_key.p, buf_p, 10) ;
  mpz_set_str(&private_key.q, buf_q, 10) ;
  /*mpz_out_str(stdout, 10, &private_key.q);*/
close(from);

/* Begin Decode using private_key */
  rsa_private(&aux, &test, &private_key);

/* convert to string */
```

```
  mpz_get_str(hex,16,&aux);
  bytes = strlen(hex);

  for (i = 0; i < bytes; i+=2) {
       str2hex(hex[i],&bytehi);
       bytehi = bytehi << 4;
       str2hex(hex[i+1],&bytelo);
       byte = bytehi | bytelo;
       sprintf(bytestr,"%c",byte);
       strncat(ret,bytestr,1);
  }
printf("return: %s\n",ret);
       return(ret);
}
```

VITA

Passakon Prathombutr

Candidate for the Degree of

Master of Science

Thesis: SECURE LOGIN OVER TCP/IP USING PUBLIC-KEY CRYPTOSYSTEM

Major Field: Computer Science

Biographical:

Education: Graduated from Nakhonsawan school, Nakhonsawan, Thailand in March 1986; received Bachelor of Science degree in Physics from Chiangmai University, Chiangmai, Thailand in February 1990 and a Master of Science degree in Computer Science from Chulalongkorn University, Bangkok, Thailand in January 1993. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1998.

Experience: Employed as a system administrator by Internet Service Office, Chulalongkorn University, Bangkok, Thailand, 1992 to 1993; Employed as a researcher by National Electronics and Computer Technology Center, Ministry of Science Technology and Environment, Bangkok, Thailand, 1993 to present.