Talking Securely

Mark Arnold
Anthony Boyd
Susan Dalton
Flora Lo
Adam Millard
Shalini Shah

November 1993
Revised February 94

Abstract

The report describes technical details of "sectalk", a security-enhanced version of the Unix "talk" program.

Talking Securely
User Manual
Software Project 1993
Group 18

Mark Arnold
Anthony Boyd
Susan Dalton
Flora Lo
Adam Millard
Shalini Shah

Supervisor : Yuliang Zheng

November 1993
Revised February 94

Contents

1 Introduction

Man was thrilled by the invention of the telephone but he didn't stop there, technology advanced and times changed. Now we find ourselves moving closer and closer into the information age. The once sedate concept of a single black telephone has been replaced by demand for more advanced communication services and we have to progress with the ever changing times. In conjunction with the ever increasing breakthroughs in communication and technology, there is also a need to maintain the privacy of individuals. In order to do this, we must ensure that the individual's rights are upheld and the protection of these rights are incorporated into these new technologies.

Current privacy laws cannot keep pace with technology that is continually being developed and changed. An individual must seek to protect their privacy, rather than rely on law makers to develop specific legislation that may be superseded by new technological developments. An individual has the right to freedom of speech, freedom of communication and the right to privacy. All of these rights are constantly being challenged by new technologies eg.monitoring of telecommunications by satellites. With this knowledge, the planning and design of a data processing network should provide safeguards so that no one can utilise the communication facilities to obtain sensitive information being transmitted through the network. Talking Securely is one such safeguard.

Prior to Talking Securely, none of an individuals rights were catered for when communicating over the network. By providing an authentication and security option, Talking Securely allows individuals to exercise their rights to communicate their thoughts without restriction.

The Talking Securely system has many applications, eg. from corporate enterprises, to students who want to communicate with each other in private with the confidence that any intercepted messages would be meaningless.

Unlike a telephone that relies on voice recognition or trust, Talking Securely enables you to be certain that the person you are communicating with, is the person you think you are communicating with via the authentication option. By using the security option of Talking Securely, individuals can protect their messages from interception and deciphering which would be difficult to ensure with telephone conversations.

The Talking Securely system implements security algorithms recently designed by J. Pieprzyk, J. Seberry and Y. Zheng (well known authorities in the security area). All algorithms in the Talking Securely system have been available in the public domain for at least one year and are yet to be successfully attacked. For these reasons, users of the Talking Securely system can feel confident that their privacy is protected every time they use the system. Talking Securely is a system that takes a good product 'talk' and makes it better by providing protection for its users.

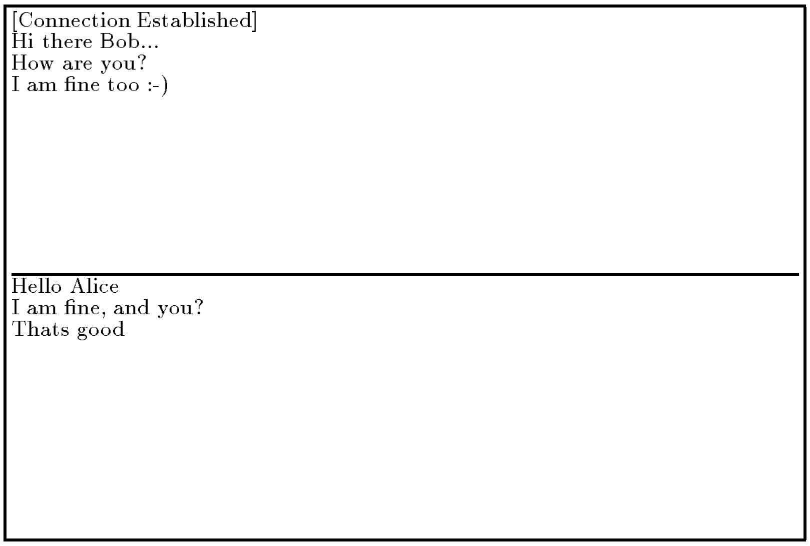

Figure 1: Screen Display Seen During 'talk'

2 A General Discussion of Talking Securely

2.1 Introduction

The aim of Talking Securely is to provide the users of the 'talk' facility on the UNIX system with security. It does this by providing users with additional options that allow authentication and security.

The 'talk' facility currently available on UNIX was developed by the University of California, Berkeley and provides a fast method of 'instant' communication across the UNIX system. To use the 'talk' facility, a user enters 'talk username [ttyname]'

eg. talk bob@wraith.cs.uow.edu.au

or

eg. talk bob@wraith.cs.uow.edu.au tty02

The machine then locates the person that the communication was requested with. If they are connected to the system, then they are prompted with a request message. If they wish to communicate with the person that sent the message then they enter 'talk username [ttyname]' (using the username of the person that sent the request)

eg. talk alice@wraith.cs.uow.edu.au

or

eg. talk alice@wraith.cs.uow.edu.au tty02

If a connection is established, then each user's screen is split in two horizontally with one user's typing appearing in the top half of the screen and the other user's typing appearing in the bottom half of the screen as seen in figure 2.1

The problem with the 'talk' facility is that it provides no protection for its users against impostors or the interception of messages. Talking Securely solves this problem.

```
[                                                 ]
[  SSS    EEEE   CCCC  TTTTT    A    L      K   K ]
[ S   S  E      C         T     A A   L      K  K  ]
[ S      E      C         T    A   A  L      K K   ]
[   S    EEEE   C         T    AAAAA  L      KK    ]
[     S  E      C         T    A   A  L      K K   ]
[ S   S  E      C         T    A   A  L      K  K  ]
[  SSS    EEEE   CCCC     T    A   A  LLLLL  K   K ]
[                                                 ]
     [***************************************]
     [*           Sectalk Menu              *]
     [*up/down to move, hit return to select*]
     [***************************************]
     [*    (N)ormal talk                    *]
     [*    (L)ow authentication             *]
     ->    (H)igh authentication            *]
     [*    (F)ull security                  *]
     [*    (Q)uit                           *]
     [*    High level of authentication     *]
     [***************************************]
```

Figure 2: Menu Display of the Talking Securely system

Talking Securely provides the users of the ‘talk’ facility with additional options that enable them to protect themselves against attack. To protect against impostors, a user can use the authentication option of Talking Securely. This option enables the user to check that they are actually communicating with the person they expect to be communicating with. To protect their communications and the privacy of their conversation, users can use the security option of Talking Securely. This option authenticates the users first and then encrypts messages before they are sent across the communication channels. These options transform the ‘talk’ facility available on UNIX into a secure system that a user can feel confident using.

2.2 About The Talking Securely System

The Talking Securely system is based on the University of California’s ‘talk’ system. The basic method of establishing connections between machines is done by the University’s code. None of the basic functioning of the University’s code has been altered by the Talking Securely group. Within the University’s code, calls have been made to functions that enable the authentication and security processes to be invoked if these options have been selected by the users of the Talking Securely system. The method of running the ‘talk’ facility has been changed slightly in the Talking Securely system - a user must use ‘sectalk’ instead of ‘talk’ to run the process. eg. ‘talk username’ becomes ‘sectalk username’.

2.3 Using The Talking Securely System

To use the Talking Securely system a user enters ‘sectalk username [tty]’. A menu will appear displaying all the options available to the user as seen in figure 2.3 The user simply need move the cursor to the option they wish to use and press return.

Alternatively, the Talking Securely system provides the user with a method of running the system without the menu via the provision of command line options. To run the system without the menu appearing, the user simply enters :

'sectalk username option' eg. 'sectalk bob -n'

The options available to a user are :

-n Normal Talk : This option provides the user with the basic sectalk session, without any authentication or security.

-a1 Low Authentication : This option provides the user with a basic checking system to authenticate the person they wish to communicate. It is the lowest level of security available on the system.

-a2 High Authentication : This option provides the user with a more secure system to authenticate the person they wish to communicate with. This option prompts the user for their private key and uses the key value entered in subsequent calculations. This option provides an addition level of checking and a more secure system.

-s Full Security : This option provides the user with the highest level of security. Duel authentication is performed with both users prompted for their private keys. If authentication is successful and a connection is established, then messages are encrypted prior to being sent across the communication channels.

It is up to the users of the Talking Securely system to determine how secure their communications need to be and then select which of the available options they should use. The system then responds to the users choices in the following way :

User A	User B	system response
-n	-n	Normal sectalk session without authentication or security.
-a1	-n	User A authenticates user B by reading private and public keys from a file.
-a2	-n	User A authenticates user B. Both users are requested to enter their private key, and the keys entered are used in subsequent calculations.
-s	-n	Duel authentication - user A authenticates user B and user B authenticates user A. Both users are requested to enter their private key, and the keys entered are used in subsequent calculations. If authentication is successful the messages are encrypted prior to being sent across the communication channels
-n	-a1	User B authenticates user A by reading private and public keys from a file.
-a1	-a1	User B authenticates user A, and user A authenticates user B, by reading private and public keys from a file.
-a2	-a1	Duel authentication, with the higher level of authentication -a2 being used by both users.
-s	-a1	Duel authentication - user A authenticates user B and user B authenticates user A. Both users are requested to enter their private key, and the keys entered are used in subsequent calculations. If authentication is successful, then messages are encrypted prior to being sent across the communication channels.
-n	-a2	User B authenticates user A. Both users are requested to enter their private key, and the keys entered are used in subsequent calculations.
-a1	-a2	Duel authentication, with the higher level of authentication -a2 being used by both users.
-a2	-a2	Duel authentication, with the higher level of authentication -a2 being used by both users.
-s	-a2	Duel authentication, with the higher level of authentication -a2 being used by both users. If authentication is successful, then messages are encrypted prior to being sent across the communication channels
-n	-s	Duel authentication, with the higher level of authentication -a2 being used by both users. If authentication is successful, then messages are encrypted prior to being sent across the communication channels.
-a1	-s	Duel authentication, with the higher level of authentication -a2 being used by both users. If authentication is successful, then messages are encrypted prior to being sent across the communication channels.
-a2	-s	Duel authentication, with the higher level of authentication -a2 being used by both users. If authentication is successful, then messages are encrypted prior to being sent across the communication channels.
-s	-s	Duel authentication, with the higher level of authentication -a2 being used by both users. If authentication is successful, then messages are encrypted prior to being sent across the communication channels.

2.4 Using Authentication

To use the Talking Securely authentication process, a user must select one of the authentication options (see section 2.3). The algorithms used in the authentication process are shown in figure 2.4

These algorithms for authentication can be found in [Zhe94]

The general process of authentication involves message comparison. In the Talking Securely system, the basic message contains a 64 bit classical key (to be used if the security process has been selected) and some 'handshaking' information. The basic message along with a combination of our private key, the other parties public key and the random number x is processed using the one-way hashing algorithm Haval [ZPS93] to produce a specific length and a digest equivalent to the compressed input message. This digest is appended to the processed input to produce a final message. Before being sent, the final message is exclusive-ORed with z to create c_2 the second part of the cipher text to be sent. The value z is a large random bit string that is created using a generating function G. The seed for G is the variable r, which consists of the other users public key raised to the power of $(x + ourprivatekey) : y_b(x + x_a)$. Note that our private and public key pair is x_a, y_a and the other users key pair is x_b, y_b.

The value x is a large random number in a specified range, this being 1 to $p - 1$.

The first part of the cipher text c_1 is created by raising g to the power of x. The value g is a primitive root in the range 1 to $p - 1$.

The value p is a large prime number (approximately 512 bits) used during the authentication process.

Once c_1 and c_2 have been created (according to Algorithm 1 - see figure 2.4), they are concatenated and sent to the other user.

When the other user receives c_1 and c_2, Algorithm 2 of the authentication process is used.

Firstly a value for r' is created to be used as the input when generating z'. If the keys used in Algorithm 1 to create the cipher texts were correct, then the value for z' should be the same as the value for z used in Algorithm 1 since we are using the same seed for the G function to create the random bit string.

$$y_a = g^{x_a}, \qquad y_b = g^{x_b}, \qquad c_1 = g^x.$$

In Algorithm 1, the seed used for the G function is :

$$\begin{aligned} y_a(x + x_b) &= gx_a(x + x_b). \\ &= gx_a x + x_a x_b. \end{aligned}$$

In Algorithm 2, the seed used for the G function is :

$$\begin{aligned} (y_b.c_1)x_a &= (gx.gx_b)x_a. \\ &= gx_a x + x_a x_b. \end{aligned}$$

Once z' is created, it is exclusive-ORed with the value of c_2 that was sent. This allows the basic message and the digest to be extracted and used. The digest is saved and will be used to confirm authentication. The basic message is reprocessed using Haval to create a new digest. For authentication to succeed, the digest sent

Algorithm 1 $E_{owh}(x_B, y_A, p, g, m)$

1. $x \in_R [1, p-1]$.
2. $r = y_A^{x_B + x} \bmod p$.
3. $z = G(r)_{[1 \cdots (P+\ell)]}$.
4. $t = h(m \mid\mid r)$.
5. $c_1 = g^x \bmod p$.
6. $c_2 = z \oplus (m \mid\mid t)$.
7. output (c_1, c_2).

end

Algorithm 2 $D_{owh}(x_A, y_B, p, g, c_1, c_2)$

1. $r' = (y_B \cdot c_1)^{x_A} \bmod p$.
2. $z' = G(r')_{[1 \cdots (P+\ell)]}$.
3. $w = z' \oplus c_2$.
4. $m' = w_{[1 \cdots P]}$.
5. $t' = w_{[(P+1) \cdots (P+\ell)]}$.
6. if $h(m' \mid\mid r') = t'$ then
   output (m')
   else
   output $(\emptyset)$.

end

Figure 3: Authentication Algorithms

and the new digest created must be the same. If the wrong key values are used (ie an impostor), or the message is corrupted during the transmission stage then authentication will fail.

If authentication fails, the connection is automatically closed. If authentication is successful, then users can be sure that they are communicating with the person the think they are communicating with.

2.4.1 The Haval Function

A lot of work has been done on designing a practical one-way hashing algorithm. Most of the algorithms designed output digests of fixed length eg. the MD family (MD2, MD4, MD5) output digests of 128 bits while the digests of SHS are 160 bits. The problem with hashing algorithms that produce a fixed length digest is that they are limited to applications that require digests of that size. Also the level of security is limited by the size of the digest - the smaller the digest the lower the level of security. To increase the usefulness of a one-way hashing algorithm, it would need to produce digests of variable lengths - this is exactly what Haval does. Haval is a one-way hashing algorithm that can produce digests of 128, 160, 192, 224 and 256 bits.

The Haval algorithm uses some of the ideas behind the design of the MD5 algorithm. It also uses five boolean functions as presented in [SZ93]. Each of the functions have the following properties which are considered desirable for a one-way hashing algorithm :

1. The functions are 0-1 balanced.
2. They are highly non-linear.
3. They satisfy the Strict Avalanche Criterion.
4. They cannot be transformed into one another by applying linear transformations to the input coordinates.
5. They are not mutually correlated via linear functions or via bias in the output.

The Haval algorithm apart from providing variable length outputs, also provides three levels of security within the algorithm : Low, Medium and High.

- In Low, Pass = 3 and each 1024 bit block of a message is processed three times to produce the digest.
- In Medium, Pass = 4 and each 1024 bit block of a message is processed four times to produce the digest.
- In High, Pass = 5 and each 1024 bit block of a message is processed five times to produce the digest.

These levels provide additional flexibility for the users of the Haval algorithm - with Haval providing a user with 15 different levels of security to choose from.

To process a message , Haval pads the message so that its length is a multiple of 1024. Part of the padding added to the message includes information about the length of the original message, the level of security being used (Pass = 3, Pass = 4 or Pass = 5), the digest length required and the version of Haval being used (currently 1). Each 1024 bit block of the padded message is then processed to produce a 256 bit digest that is equivalent to the compressed message (according to the value of Pass used). The 256 bit digest produced is then tailored to produce a digest of the size requested by the user (128, 160, 192, 224 or 256).

With respect to other one-way hashing algorithms, Haval is regarded as better in terms of speed and security. The Haval algorithm is more secure than the MD5 algorithm, which in turn is considered faster and more secure compared with other one-way hashing algorithms. Experiments done by the developers of the Haval algorithm found that :

- When 3 passes were used, Haval was at least 60% faster than MD5.
- When 4 passes were used, Haval was at least 15% faster than MD5.
- When 5 passes were used, Haval was as fast as MD5.

2.5 Security

The security option of the Talking Securely system provides users with the highest level of authentication as well as the encryption/decryption of messages.

If a user selects the security option, the system automatically sets the options for both users to 'security'. Duel authentication occurs, with user A creating a message and authenticating user B, while B creates a message and authenticating user A. Both users are prompted for their private keys, and the keys entered are used in the authentication calculations. If authentication is successful, then all messages are encrypted prior to being sent across the communication channels.

The Classical Key method is used in the Talking Securely system. With the classical key method, the same key is used by both users to encrypt and decrypt messages. The classical key is distributed by making it part of the basic message used in the authentication process. The key is safe since the message is exclusive-ORed with a random bit string before it is sent across the communication channels (see section 2.4).

Since each user generates a message, then they also generate a version of the key. Each users key is extracted from the corresponding message and the two keys are exclusive-ORed to produce a final classical key which will be used in the encryption/decryption process (see figure 2.5).

The characters typed to the screen are stored in a communication buffer. The communication buffer is then processed and encrypted using the a method of exclusive-ORing the data in the buffer with a random bitstring produced by a random string generator. In this case the routines used to generate the random bit string are found

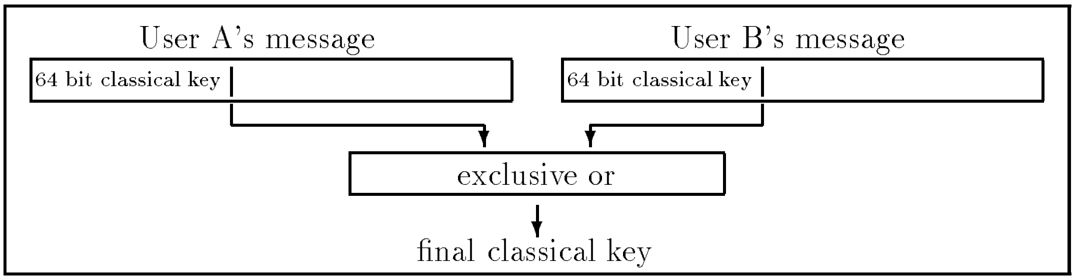

Figure 4: Process of Creating the Final Classical Key

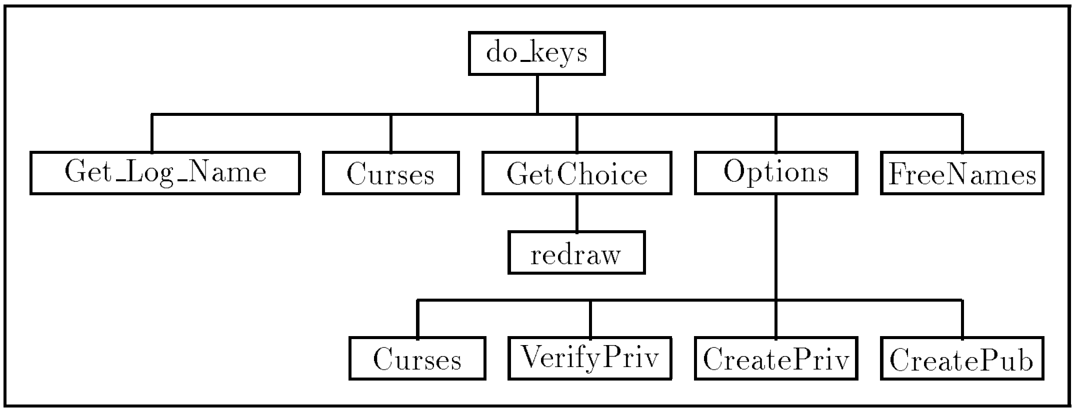

Figure 5: Module hierarchy for Do_Keys.

in the STRANDOM [Zhe93] library. STRANDOM which stands for STrong RANDOM number gererator is a useful package containing a number generator based on Haval.

The communication buffer is then sent to the other user. The encrypted characters in the communication buffer are then processed and decrypted again using the exclusive-OR method.

All valid characters will appear on the user's screen when the calling function displays them.

3 A Detailed Discussion of Talking Securely

3.1 Creating a Set of Keys

Before a user can run the Talking securely system, they must create a pair of keys - a public key and a private key. To create these keys, a user must run the 'pubkey' program. When this program is run, a call is made to the 'Do_Keys' function, which calls all necessary functions to create a pair of keys for the user (see figure3.1). Once the keys are created, they are placed into files, ready to be used by the system. The 'Do_Keys' function consists of a main line that calls the functions required to create the keys.

NAME OF MODULE : do_keys

```
[*************************************]
[        Key Generation Program       ]
[*************************************]
[*************************************]
[*->  (N)ew set of keys              *]
[*    (Q)uit                         *]
[*                                   *]
[*    Create a new set of keys       *]
[*************************************]
```

Figure 6: Key Generating Menu

DESIGNED BY : Anthony Boyd, Flora Lo, Adam Millard, Shalini Shah
DEVELOPED BY : Flora Lo
WRITTEN BY : Flora Lo
TESTED BY : Flora Lo
VARIABLES

int i - Loop index variable.
int j - Array index variable.
static char heading[9][40] - For storing the key generation program title and the menu options.

ABOUT THE ROUTINE - Mainline do_keys

This routine initially sets up the communicating parties login names, after which calls are made to various curses routines to set up the window display. A new window is created and the program title along with the menu options are displayed, using a preinitialised array. This generates a screen display as seen in figure3.1. This is followed by calls to routines to obtain the users selection and perform the desired function. Finally calls are made to terminate the window display and release the users names.

NAME OF MODULE : GetLogName
DESIGNED BY : Mark Arnold and Adam Millard.
DEVELOPED BY : Mark Arnold and Adam Millard.
WRITTEN BY : Mark Arnold and Adam Millard.
TESTED BY : Mark Arnold and Adam Millard.

FUNCTION PROTOTYPE

void
GetLogName ();

PARAMETERS

This function takes no parameters.

VARIABLES

name - This is the login name of the user.

ABOUT THE FUNCTION

The users login name is found by the function getlogin (), but if this fails (ie. returns a NULL) then it is attempted to be found by the command getpwuid (getuid)), which has as one of its fields the users login name.

Once the users login name is established then the name for the public and private key files are then created. This is done by combining the users login name with a prefix (which is the path to the directory in which the users keys are to be found) and an extension (either ".pub" or ".prv", depending on which type the key file is).

The variables pubname and privname are globals.

CURSES

The CURSES functions are standard libraries that are available on the UNIX system. It is a screen display package. To use curses, it is a matter of including the header file in functions that do any screen displays, and compiling the function appropriately.

NAME OF MODULE: get_choice.
DESIGNED BY : Flora Lo.
DEVELOPED BY : Flora Lo, Adam Millard.
WRITTEN BY : Mark Arnold, Flora Lo.
TESTED BY : All Members.

FUNCTION PROTOTYPE

char
get_choice();

PARAMETERS

No parameters passed in.

VARIABLES

unsigned in_char - Key entered by the user.
int i - Loop variable for updating the arrow of the current option with respect to user's responds.
int top_line - Stores the line of the first option on the menu.

int bottom_line - Stores the line of the last option on the menu.
static char display_selec[2][40] - Stores the the explanation of the current option on the bottom of the menu.

ABOUT THE FUNCTION

Firstly the variables i, top_line, bottom_line are initialised. With i = 4 this indicates that the current option is normal sectalk, top_line = 4 which indicates that the first option on menu is the top boundary, and finally the bottom_line = 5 which indicates that the last option is the bottom boundary.

The program waits for a user to select an option. When an option is selected, the program reads in the key entered in by the user from keyboard. It then modifies the screen depending on the value of the key entered. If the key entered is KEY_DOWN, the function redraw is called to clear the arrow at the current position. If the value of i is less than last option, i is then incremented (indication of scrolling to next option), otherwise i is set the top option.

If the key entered is KEY_UP,The function redraw is called to clear the arrow at the current position. If the value of i is greater than top option, i is then decremented (indication of scrolling up to the previous option), otherwise i is set to the last option.

Otherwise, check if the key entered is equal to one of the valid character that represents the particular option.

If the key is 'n' or 'N' : The function redraw is called to clear the arrow at current position. Then i is set to 4 (indication of first option which is new keys).

If the key is 'q' or 'Q' : The function redraw is called to clear the arrow at current position. Then i is set to 4 (indication of quit).

Each time a user moves the cursor, an arrow is displayed at the ith option, and an explanation of the current option is displayed at the bottom of the option menu.

If the key entered is KEY_ENTER (indication of selecting the current option), then the function checks the value of i which represents the ith option of the menu.

If i is 4 : return the character 'N' to the calling function (indicating a new set of keys is to be created).

If i is 5 : return the character 'Q' to the calling function (indicating the termination of the process).

NAME OF MODULE : Options
DESIGNED BY : Adam Millard.
DEVELOPED BY : Adam Millard.
WRITTEN BY : Adam Millard.
TESTED BY : Adam Millard.

FUNCTION PROTOTYPE

void
Options (choice)

char choice;

PARAMETERS

choice - This is the choice made by the user at the menu.

VARIABLES

g - This is the random number used in the generation of the users public key.
prime - This is the prime number used in the generation of the users public key.
p_and_g - This is the pointer to the file which contains the prime and random numbers for the generation of the public key.
ch - This is a pointer to the character string containing the users password
ver - This integer variable is used as an indication of the result when the users key is verified

ABOUT THE FUNCTION

This function clears and refreshes the window and then acts on the choice of the user (from the menu) in a switch statement.

If the users choice is 'N' or 'n' then a call to VerifyPriv () is made which is used to verify the users password as entered from the keyboard with that in the file if such a private key file exists. A call is then made to GetPriv to read in the new password from the keyboard if necessary. The CreatePriv routine is then processes the password to create the private key storing it in the appropriate private key file.

If this is successful then the file containing the random and prime numbers is opened and the random and prime numbers are read in. If this file could not be opened for some reason then the user is prompted to enter these in manually.

After this a call is made to the function CreatePub (), which creates the public key and file from the users private key, random number read in, and the prime number read in.

Otherwise, if the users choice is 'Q' or 'q' then the screen is cleared and refreshed and the function returns without altering the public or private key files.

NAME OF MODULE : CreatePub
DESIGNED BY : Adam Millard.
DEVELOPED BY : Mark Arnold and Adam Millard.
WRITTEN BY : Mark Arnold and Adam Millard.
TESTED BY : Mark Arnold and Adam Millard.

FUNCTION PROTOTYPE

```
int
CreatePub (g, prime)
long **g;
long **prime;
```

PARAMETERS

g - The random number used in the generation of the users public key.
prime - The prime number used in the generation of the users public key.

VARIABLES

pubkey - This is the users public key.
privkey - This is the users private key.
pubkey_file - This is the pointer to the users public key file.

ABOUT THE FUNCTION

If the public key file could be opened/created then the users public key is created from their private key using the function ConvertPriv (). Then the users public key, the random number (g) used in its creation, and the prime number used to generate the public key are then written to the public key file using the function zfwrite(), from the lentstra_2 large numbers package.

If the public key is successfully created then the function returns '1' to the calling function, otherwise it returns '-1'.

NAME OF MODULE : CreatePriv.
DESIGNED BY : Adam Millard.
DEVELPOED BY : Adam Millard.
WRITTEN BY : Adam Millard.
TESTED BY : Adam Millard.

FUNCTION PROTOTYPE

```
int
CreatePriv ();
char *ch;
```

PARAMETERS

ch - This is the users password which is used to generate the private key

VARIABLES

privkey_file - This is a pointer to the private key file.
privkey - This is the new private key typed in by the user.

ABOUT THE FUNCTION

It is assumed that the user has already entered their password prior to calling this routine and this password is passed in as the parameter 'ch'.

A call is made to the routine Set_Seed() to seed the random number generator used in the secret key generation, with the input being the users password. The routine gen_secretkey() is used to generate the key which returns a pointer to the private key.

Noe that the key has been generated and attempt is made to generate open or create the users private key file.

The new private key is then encrypted using the routine EncryptPriv() which also writes to the private key file, overwriting any existing private key.

This function returns a value of 1 to the calling function if the private key was successfully changed, and -1 if the it was not.

NAME OF MODULE : FreeNames
DESIGNED BY : Adam Millard.
DEVELOPED BY : Adam Millard.
WRITTEN BY : Adam Millard.
TESTED BY : Adam Millard.

FUNCTION PROTOTYPE

```
void
FreeNames ();
```

PARAMETERS

This function takes no parameters.

VARIABLES

This function has no variables.

ABOUT THE FUNCTION

This function frees the memory which was allocated to the global variables pubname and privname. This memory was allocated in the function GetLogName ().

3.2 The User Interface

Once the system is run, the system calls the function check_options. If no valid options are found on the command line, then the Do_Menu function is called. Do_menu displays all the options available to the user, and accepts and acts on their choice. (see section 2.1) See figure 3.2 for module hierarchy.

NAME OF MODULE : check_options.
DESIGNED BY : Mark Arnold.
DEVELOPED BY : Mark Arnold and Anthony Boyd.

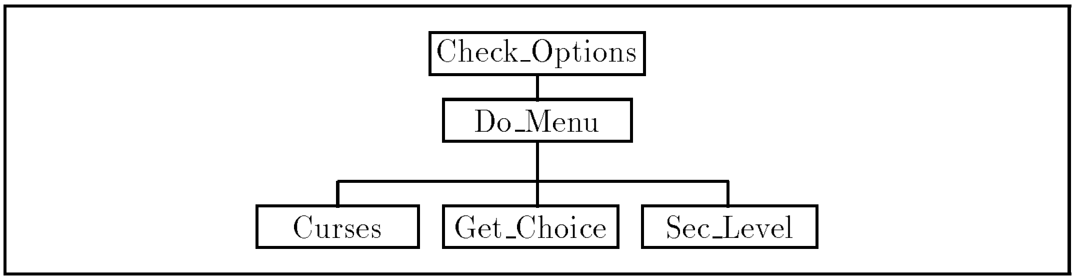

Figure 7: Module hierarchy for main User Interface

WRITTEN BY : Anthony Boyd and Mark Arnold.
TESTED BY : Mark Arnold and Anthony Boyd.

FUNCTION PROTOTYPE

```
check_options(argc,argv)
int             argc;
char            *argv;
```

PARAMETERS.

The parameters passed to this function are the command line arguments that the user typed on the command line, these being :
argc : This is the total number of arguments that the user had typed.
argv : This is an array of strings 'argc' in length containing pointers to the typed arguments.

VARIABLES.

i : This is a loop variable which is used as an index when scanning the arguments for the appropriate options.
menuoption : This is a boolean value which determines if the menu needs to be displayed or not.
It is used in case the user wishes to use the menu and has also placed a tty in the command line arguments increasing the number of arguments by one and using up a place in the options.

ABOUT THE PROCEDURE

This routine is called after the get_names procedure has processed the command line data referring to the party that the user wishes to communicate with. This procedure searches the command line for the options '-s', or '-a2', or '-a1', or '-n' and takes the appropriate action, this being the setting of specific global boolean flags.

If no such options are found on the command line, the procedure calls another procedure do_menu which displays a menu for the user to enter the options that they want there.

The body of the procedure is a for loop which executes over all elements in the argv array except argv[0] which is the program name. In this loop string comparisons are made against '-n', '-a1', '-a2' and '-s', and if any of these are found the appropriate flags are set as explained above.

As well as checking for options on the command line, this module also contains the declarations for some of the global variables.

The first entry in this file is that of a type definition. This definition is the standard boolean definition typedef enum false, true boolean; which defines false to 0 and true to 1. Following this are the declaration of five boolean flags, these being :

boolean	authenticate1 = false;
boolean	authenticate2 = false;
boolean	security = false;
boolean	duel = false;
boolean	initiator = false;

All these flags are initially set to false and are set according to the following conditions :
The authenticate1 flag is set when the user specifies '-a1' as an option on the command line.
The authenticate2 flag is set when the user specifies '-a2' as an option on the command line, or the '-s' option is there.
The security flag will be set when the '-s' option is found on the command line, and indicates that the user requires the conversation between parties to be encrypted when sent over the communication channel.
The duel flag is merely initialised here. It will be set when both parties wish to authenticate each other or the security option is chosen.
The initiator flag is set when either of the authenticate options are chosen and is used to determine who the authenticating party is.

NAME OF MODULE : do_menu.
DESIGNED BY : All Members.
DEVELOPED BY : Flora Lo.
WRITTEN BY : Flora Lo.
TESTED BY : Mark Arnold.

FUNCTION PROTOTYPE

```
void
do_menu();
```

PARAMETERS
No parameters passed in this function.

VARIABLES

```
[                                                ]
[  SSS    EEEE   CCCC  TTTTT   A    L      K   K ]
[ S   S  E      C        T    A A   L      K  K  ]
[ S      E      C        T   A   A  L      K K   ]
[   S    EEEE   C        T   AAAAA  L      KK    ]
[     S  E      C        T   A   A  L      K K   ]
[ S   S  E      C        T   A   A  L      K  K  ]
[  SSS    EEEE   CCCC    T   A   A  LLLLL  K   K ]
[                                                ]
     [***************************************]
     [*          Sectalk Menu               *]
     [*up/down to move, hit return to select*]
     [***************************************]
     [*    (N)ormal talk                    *]
     [*    (L)ow authentication             *]
     ->    (H)igh authentication            *]
     [*    (F)ull security                  *]
     [*    (Q)uit                           *]
     [*    High level of authentication     *]
     [***************************************]
```

Figure 8: Main Menu Display

int	centre	- is for storing the middle column of the screen.
int	i	- Loop index variable.
static char	talk_heading[9][50] - For storing the sectalk title.	
static char	menu_option[12][40] - For storing the menu options and menuheading.	

ABOUT THE PROCEDURE

This procedure calls different curses routines to set up the screen. Two windows are defined for the option menu, large_win, menu_win.

These windows are defined separately for ease of modification. The function first creates and initialises the windows, the option menu and 'sectalk' title being displayed on screen simply by printing out preinitialised arrays. The two dimensional array talk_heading[9][50] is initialised to the main menu display (see figure 3.2). For loops are used for displaying each initialised array in a defined window. Talk_heading is displayed in the window large_win and menu_option on menu_win window.

The function get_choice is then called, it provides the routines to get the user's response and then determines the user's choice of option. It then returns a character which represents the user's chosen option. Finally, do_menu calls the function Sec_level which sets appropriate flags for authentication and security process depending on the option chosen.

NAME OF MODULE : get_choice.
DESIGNED BY : Flora Lo.
DEVELOPED BY : Flora Lo.
WRITTEN BY : Mark Arnold and Flora Lo.
TESTED BY : Mark Arnold.

FUNCTION PROTOTYPE

char
get_choice();

PARAMETERS

No parameters passed in.

VARIABLES

The variables used in the this function are :

unsigned long in_char - Key entered by the user.
int i - Loop variable for updating the arrow pointing to the current option with respect to user's responds.
int top_line - Stores the line of the first option on the menu.
int bottom_line - Stores the line number of the last option on the menu.
static char display_selec[5][40] - Stores the explanation of the current option on the bottom of the menu.

ABOUT THE PROCEDURE

Firstly the variables i, top_line, bottom_line are initialised. With i = 4 which indicates that the current option is normal talk, top_line = 4 which indicates that the first option on menu is the top boundary, and finally the bottom_line = 8 which indicates that the last option is the bottom boundary. The function then repeatedly tries to get the user's option by :
Reading in the key entered in by the user from the keyboard, and modifying the screen depending on the value of the key entered. If the key entered is KEY_DOWN, the function redraw is called to clear the arrow at the current option. If the value of i is less than the bottom_line, i is then incremented (indication of scrolling to next option), otherwise i is set as the top_line. If the key entered is KEY_UP, the function redraw is called to clear the arrow at the current option. If the value of i is greater than the top_line, i is then decremented (indication of scrolling up to the previous option), otherwise i is set to the bottom line. Otherwise, the function checks to see if the key entered is equal to one of the valid character that represents the particular option.

If the key is 'n' or 'N' : The function redraw is called to clear the arrow at the current option. Then i is set to 4 (indication of the first option which is normal sectalk).

If the key is 'l' or 'L' : The function redraw is called to clear the arrow at the current option. Then i is set to 5 (indication of the second option which is low authentication).

If the key is ‘h’ or ‘H’ : The function redraw is called to clear the arrow at the current option. Then i is set to 6 (indication of the third option which is high authentication). If the key is ‘f’ or ‘F’ : The function redraw is called to clear the arrow at the current option. Then i is set to 7 (indication of the forth option which is full security ie. high authentication plus security).

If the key is ‘q’ or ‘Q’ : The function redraw is called to clear the arrow at the current option. Then i is set to 8 (indication of quit).

An arrow is displayed at the *i*th option. The explanation of the current option is displayed at the bottom of the option menu. If the key entered is KEY_ENTER (indication of selecting the current option). Checks on the value of i which represent the *i*th option of the menu.

If i is 4 : return the character ’N’ to do_menu.
If i is 5 : return the character ’L’ to do_menu.
If i is 6 : return the character ’H’ to do_menu.
If i is 7 : return the character ’F’ to do_menu.
If i is 8 : return the character ’Q’ to do_menu.

NAME OF MODULE : redraw.
DESIGNED BY : Flora Lo.
DEVELOPED BY : Mark Arnold and Flora Lo.
WRITTEN BY : Mark Arnold and Flora Lo.
TESTED BY : Mark Arnold and Flora Lo.

FUNCTION PROTOTYPE

```
void
redraw(i)
int i;
```

PARAMETERS

i - is used to indicate which option the arrow cursor is currently displayed at.

VARIABLES

No variables used.

ABOUT THE PROCEDURE

Move the cursor to the *i*th option of the menu. Display two spaces at the position of the arrow on the screen to implement as clearing of the arrow at the *i*th option.

NAME OF MODULE : Sec_level
DESIGNED BY : Flora Lo.
DEVELOPED BY : Flora Lo.

WRITTEN BY : Anthony Boyd, Flora Lo, Adam Millard, Shalini Shah.
TESTED BY : Mark Arnold.

FUNCTION PROTOTYPE

```
void
Sec_level(choice)
char choice;
```

PARAMETERS

choice - contains the option selected by the user.

VARIABLES

No variables used.

ABOUT THE PROCEDURE

This function checks on the value stored in choice. It then sets flags for authentication and security process depending on the value of choice.

If choice is 'N' : No flags are set and the function simply returns

If choice is 'L' : Authenticate1, and initiator are set to true;

If choice is 'H' : Authenticate2, and initiator are set to true;

If choice is 'F' : Authenticate2, security and initiator are set to true;

If choice is 'Q' : No flags are set, the windows are cleared and the quit() routine is called.

3.3 The Authentication Process

The authentication process is invoked when a user selects an authentication or security option. The main line procedure in talk.c calls the authentication process after all initial connection procedures are completed (display initialised, sockets opened and edit characters are set and sent).

The authentication process involves getting user information and keys, creating a message and performing the required sections of the authentication algorithms (described in section 2.4). See Figure 3.3 for a module hierarchy of the authentication process.

NAME OF MODULE : authentication
DEVELOPED BY : Mark Arnold

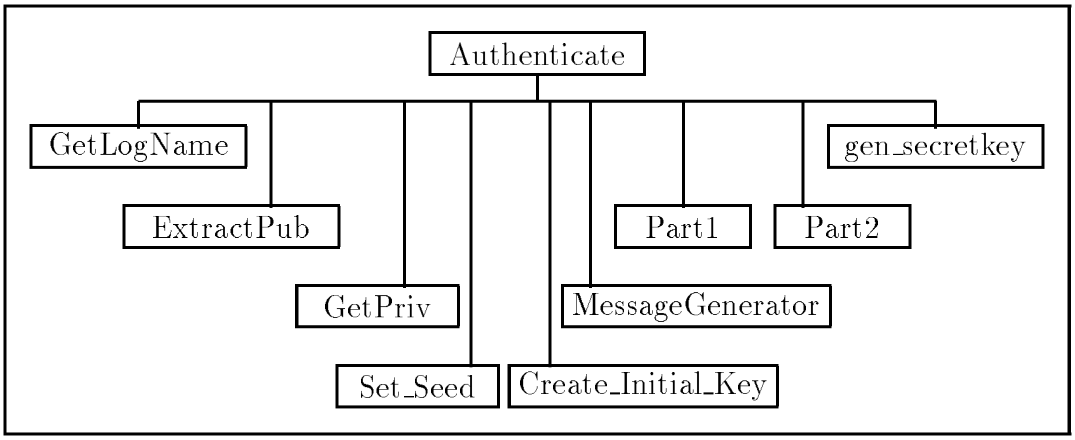

Figure 9: Module Hierarchy of Authentication Process

WRITTEN BY : Mark Arnold, Anthony Boyd
TESTED BY : Mark Arnold, Anthony Boyd, Susan Dalton

FUNCTION PROTOTYPE

void
authentication()

PARAMETERS

This function requires no parameters and calls the main authentication routines.

VARIABLES.

i,s	: These are loop variables.
fptlen	: This is the digest length produced by Haval.
cc	: An integer variable that is used when writing and reading the buffer between communicating parties to determine if some of the buffer has been lost and hence the status of the connection.
success	: This boolean variable is set when the routine Part2 is called (seebelow) and is an indication of whether authentication was successful or not.
ya	: This is a pointer to the users public key.
g	: This is a pointer to the generator number which was used in the generation of the users public and private keys.
prime	: A pointer to a long, this will point to the prime which was used in the generation of the users public and private key pair.
xb	: This pointer to a long will point to the users private key upon it being obtained.

mesg	: This pointer to a character will upon calling the Message Generator point to the message to be manipulated.
pkey	: This will point to the users password upon being typed from the keyboard as a string of characters, which will be used to generate the users public key.
c1	: This is a structure used to contain the ciphertext c_1
c2	: This is a structure used to contain the ciphertext c_2

ABOUT THE PROCEDURE

This routine determines what type of authentication has to be done, and calls the appropriate procedures/functions to do the job.

Initially a call is made to display a message on the users screen to inform them that authentication has commenced, following this a call is made to GetLogName() which sets up the file names where the users public and private keys are to be found.

Depending on which authentication flag is set determines if the users private key is verified with the users private key file. In either case calls are made to the routines GetPriv() to read in the users password from the keyboard, Set_Seed() to initialise the random number generator for generation of the users private key followed by a call to gen_secret key to create the users private key itself. In the case where the private key is to be verifyed with the users stored private key a call is made to ExtractPriv() and the keys are compared. The windows are now cleared and redrawn. The next step is to generate the keys which will be inserted in the message to be compressed and used as the key for if the user wishes to encrypt/decrypt messages.

A call to GetPub() is made and the users other public key and the components used to generate it are obtained (y_b : users public key, g : the generating number used in key generation, and $prime$: a prime number used to generate the users keys).

In the case where dual authentication has been selected the procedure takes the following actions, a message is generated, this is then passed to a routine Part1() which is an implementation of algorithm 1 as described in this manual, from which a ciphertext pair is generated and sent to the other communicating party, an attempt is then made to read a ciphertext pair sent from the other party who is doing the same actions. Upon obtaining successful ciphertexts from the other party, a routine Part2() is called, the return value of which indicating whether authentication has been successful or not.

In other cases where dual authentication has not been selected, different actions are taken according to whether we initiated the secure talk session or not. If we did not initiate the authentication but it was selected then we are required to send a ciphertext pair to the other party and do so by calling the routine Part1(), after which we are temporarily connected to the other party while they complete the authentication process. In the case where we initiated the authentication status, we make an attempt to read the ciphertext pair that the other party has sent to us and upon success we complete our actions in the authentication algorithm, by calling Part2() the result of which will tell us whether or not the other party is the one we really wish to talk to. In the case where authentication has failed the connection is closed and the user informed of what has happened, else the connection is established and the user informed that authentication has been successful.

NAME OF MODULE : ExtractPub.
DESIGNED BY : Adam Millard.
DEVELOPED BY : Mark Arnold and Adam Millard.
WRITTEN BY : Mark Arnold and Adam Millard.
TESTED BY : Mark Arnold.

FUNCTION PROTOTYPE

```
int
ExtractPub (publickey, g, prime)
long **publickey;
long **g;
long **prime;
```

PARAMETERS

publickey - This is the public key of the user.
g - This is the random number used in the generation of the users public key.
prime - This is the prime number used in the generation of the users public key.

VARIABLES

FILE *pubkey_file - the pointer to the users public key file.

ABOUT THE PROCEDURE

If the public key file can be opened for reading, then this function reads the users public key, g (primitive root), and prime number (g and prime are used in the creation of the users public key) from the public key file. These are stored in the appropriate parameters, which are returned to the calling function.

The function returns '1' if the public key file could be opened for reading and '0' if it could not be opened for reading.

The function zfread() (from lenstra_2) is used to read the numbers from the public key file.

3.3.1 Creating the Key Sent in The Message

Part of the message sent in the authentication process is a 64 bit key. This key is used to create a final classical key if the security option of Talking Securely has been selected. The classical key is then used to encrypt and decrypt all plain text entered by the users in their sectalk session (see figure 3.3.1).

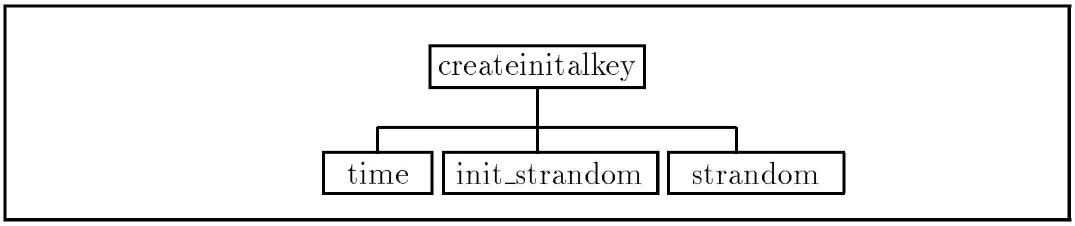

Figure 10: Module Hierarchy of CreateInitialKey

NAME OF MODULE : createinitialkey
DESIGNED BY : Anthony Boyd
DEVELOPED BY : Anthony Boyd
WRITTEN BY : Anthony Boyd
TESTED BY : Anthony Boyd

FUNCTION PROTOTYPE

void
createinitialkey()

PARAMETERS

No parameters.

VARIABLES

long seed[2] : This is the initial seed for a pseudo random Long Number generator.
time_t t : This contains the current time as a pointer to a long.
It is used as a initialiser for the seed.

ABOUT THE FUNCTION

Firstly the seed for the random number generator strandom is set using the current time, this is done through a call to init_strandom().

Two pseudo random integers are then generated by the strandom() function which sets the key that this user will send to the other user. This is one of the components used in calculating the actual classical key.

subsubsectionGetting the Private key

The way a private key is obtained depends on the type of authentication being performed. If the authentication option is -a1 low authentication, then the private key is read from a file, using the ExtractPriv function and verifyed with the key generated from the users password which has been typed in from the terminal. If the authentication option is -a2 high authentication, or the security option has been chosen, then the users are prompted for their password, by calling the GetPriv function from which a secret key will be generated.

NAME OF MODULE : ExtractPriv
DESIGNED BY : Adam Millard
DEVELOPED BY : Adam Millard.
WRITTEN BY : Adam Millard.
TESTED BY : Adam Millard.

FUNCTION PROTOTYPE :

```
long*
ExtractPriv ()
```

PARAMETERS

No parameters.
VARIABLES

FILE	*privkey_file - the pointer to the users private key file if that file could be opened.
long	*privatekey - this is a pointer to the users private key which will be returned.

ABOUT THE PROCEDURE

This function initially attempts to open the users private key file, the name of which is stored in the global variable ‘prvname’, and if successful a call will be made to the DecryptPriv() routine to read and decrypt the key from the users file. If successful the users decrypted key will be returned otherwise a key with the value of zero will be returned.

NAME OF MODULE : GetPriv
DESIGNED BY : Flora Lo
DEVELOPED BY : Flora Lo
WRITTEN BY : Flora Lo
TESTED BY :

FUNCTION PROTOTYPE

```
void
GetPriv(key, on)
char **key;
int on;
```

PARAMETERS

key - is pointer to pointer of characters. It is used to store the private key.

on - used to indicate which message to display on the screen when getting the key

VARIABLES

i,j - loop variables
win_display[8][40] used to hold the menu box displayed on the screen

ABOUT THE PROCEDURE

Calls are made to different curses routines to set up the screen. A window is then defined for screen manipulations. A menu border of asterisks is then drawn in this window using the preinitialised array win_display. Depending on the value of the variable on a version of the routine get_key() is called. If on is '0' we are simply getting the users password so get_key() is called. If on is '1' we are getting the users old password so get_old_key() is called and if on is '3' we are getting the users new password so get_new_key is called. Finally the private key is returned from the appropriate routine and it is assigned to the character pointer key.

NAME OF MODULE : get_new_key
DESIGNED BY : Flora Lo
DEVELOPED BY : Flora Lo
WRITTEN BY : Flora Lo
TESTED BY : Flora Lo

FUNCTION PROTOTYPE

```
static char*
get_new_key()
```

PARAMETERS

No parameters passed into this function.

VARIABLES

int i	- Index for password array.
char in_char	- Input character read.
char ch	- Dummy variable for displaying on screen when one character is read. It is initialised to asterisk.
boolean finish	- Loop control flag.
boolean flag	- Flag to indicate whether clearing the error message is required.
char *p_word	- Storing the password.
char password[12]	- Storing every character read in for the password excluding the enter key(KEY_ENTER).

```
[************************************]
[*                                  *]
[*  Enter your new password         *]
[*                                  *]
[*  New password: **********        *]
[*                                  *]
[*                                  *]
[************************************]
```

Figure 11: New Password Menu Display

static char sec_key_str_new[3][40] - Storing messages for prompting the private key.
static char error_msg1 - Storing the error message 1.
static char error_msg2 - Storing the error message 2.

ABOUT THE PROCEDURE

The two dimensional array sec_key_str_new[3][40] is initialised to the menu display strings to produce an appropriate menu. See Figure 3.3.1

Then the two error messages error_msg1 and error_msg2 are initialised as follows.

error_msg1[40] = "* Error : Please enter password *\0"
error_msg2[40] = "* password too long, try again! *\0"

The strings for prompting the private key are displayed inside the box of the defined window get_key_win on screen.

The function continuously tries to read a valid password (1 ¿= password ¡= 12). Every time a character other than the return key is read, it is stored in the array private_key and a dummy variable '*' is displayed on screen at the defined area in the window to show the user that a character has read. Error messages will be displayed if the password is too long (all the asterisks are cleared when this occurs), or when the user presses return without entering any characters for a password. A flag is used to determine whether it there is a error message displayed on the screen. If it is assigned to true, then the error message is displayed on the bottom of the window, the window is cleared and the flag is reset to false.

When the user presses the return key after entering their password, the characters in the password are in the range of 1 ¿= password ¡= 12, and the function does a string copy from the password array to p_word which is then returned to the calling function(GetPriv in this case).

NAME OF MODULE : get_old_key
DESIGNED BY : Flora Lo
DEVELOPED BY : Flora Lo

```
[************************************]
[*                                  *]
[*  Enter your old password         *]
[*                                  *]
[*  Old Password : **********       *]
[*                                  *]
[*                                  *]
[************************************]
```

Figure 12: Old Password Menu Display

WRITTEN BY : Flora Lo
TESTED BY : Flora Lo

FUNCTION PROTOTYPE

static char*
get_old_key()

PARAMETERS

No parameters passed into this function.

VARIABLES

int i	- Index for password array.
char in_char	- Input character read.
char ch	- Dummy variable for displaying on screen when one character is read. It is initialised to asterisk.
boolean finish	- Loop control flag.
boolean flag	- Flag to indicate whether clearing the error message is required.
char *p_word	- Storing the password.
char password[12]	- Storing every character read in for the password excluding the enter key(KEY_ENTER).
static char sec_key_str_old[3][40]	- Storing messages for prompting the private key.
static char error_msg1	- Storing the error message 1.

ABOUT THE PROCEDURE

The two dimensional array sec_key_str_old[3][40] is initialised to the menu display strings to produce an appropriate menu. See Figure 3.3.1

Then there is the error message error_msg1 which is initialised as follows.

error_msg1[40] = "* Error : Please enter password *\0"

The strings for prompting the private key are displayed inside the box of the defined window get_key_win on screen.

The function continuously tries to read a valid password (1 ¿= password ¡= 12). every time a character other than the return key is read, it is stored in the array password and a dummy variable '*' is displayed on screen at the defined area in the window to show the user that a character has read. An error message will be displayed if the user presses return without entering any characters for the password. A flag is used to determine whether it there is a error message displayed on the screen. If it is assigned to true, then the error message is displayed on the bottom of the window, the window is cleared and the flag is reset to false.

When the user presses the return key after entering their password, the characters in the password are in the range of 1 ¿= password ¡= 12, and the function does a string copy from the password array to p_word which is then returned to the calling function(GetPriv in this case).

NAME OF MODULE : get_key
DESIGNED BY : Flora Lo
DEVELOPED BY : Flora Lo
WRITTEN BY : Flora Lo
TESTED BY : Flora Lo

FUNCTION PROTOTYPE

static char*
get_key()

PARAMETERS

No parameters passed into this function.

VARIABLES

int i	- Index for private_key array.
char in_char	- Input character read.
char ch	- Dummy variable for displaying on screen when one character is read. It is initialised to asterisk.
boolean finish	- Loop control flag.
boolean flag	- Flag to indicate whether clearing the error message is required.
char *p_word	- Storing the password.
char password[12]	- Storing every character read in for the password excluding the enter key(KEY_ENTER).
static char sec_key_stri[3][40]	- Storing messages for prompting the private key.

```
[***********************************]
[*                                 *]
[*                                 *]
[*                                 *]
[*    Password : **********        *]
[*                                 *]
[*                                 *]
[***********************************]
```

Figure 13: Simple Password Menu Display

static char error_msg1 - Storing the error message 1.

ABOUT THE PROCEDURE

The two dimensional array sec_key_stri[3][40] is initialised to the menu display strings to produce an appropriate menu. See Figure 3.3.1

Then the error message error_msg1 is initialised as follows.

error_msg1[40] = "* Error : Please enter private key *\0"

The strings for prompting the private key are displayed inside the box of the defined window get_key_win on screen.

The function continuously every time to read a valid password (1 ¿= password ¡= 12). Every time a character other than the return key is read, it is stored in the array password and a dummy variable '*' is displayed on screen at the defined area in the window to show the user that a character has read. Error messages will be displayed if the user presses return without entering any character for the password. A flag is used to determine whether it there is a error message displayed on the screen. If it is assigned to true, then the error message is displayed on the bottom of the window, the window is cleared and the flag is reset to false.

When the user presses the return key after entering their private key, the characters in the private key are in the range of 1 ¿= password ¡= 12, and the function does a string copy from the password array to p_word which is then returned to the calling function(GetPriv in this case).

NAME OF MODULE : Message Generator
DESIGNED BY : Anthony Boyd
DEVELOPED BY : Anthony Boyd
WRITTEN BY : Anthony Boyd
TESTED BY : Anthony Boyd

FUNCTION PROTOTYPE

char*
MessageGenerator(key);
unsigned int *key;

PARAMETERS

key : this is part of the final classical key which will be sent to the other party.

VARIABLES

MessageGenerator, contains the variables

char *PlainText	as its name suggests is a simple string which forms part of the final message.
long *mess	- is a pointer to the final message.
time_t gentime	- the time of message creation.
char *thetime	- The string representation of the time.

ABOUT THE PROCEDURE

This function creates a message used in the authentication process. The contents of the message include a simple plain text message, the time that the message generator was called, and part of the classical key used for encryption and decryption. Since the contents of the message itself is not of prime importance in the authentication process, it did not matter what it was, however it was important that its length be kept to a minimum to avoid wasting valuable execution time.

The message string is made up of the PlainText concatenated with time. The time is assigned the string representation of the variable gentime which contains the date in the standard C format of type time_t.

The format of the message itself is a character string that gets converted to a pointer to a character. This is done for ease of storage and also for when the message is used in the actual authentication process and more specifically in the message compression function Haval.

3.3.2 Part 1

This section of code performs the first part of the authentication process, by implementing algorithm 1 of the authentication algorithms used in this system (see section 2.4). Part 1 module hierarchy is shown in (see figure 3.3.2).

NAME OF MODULE : Part1
DESIGNED BY : Y. Zheng, J. Seberry
DEVELOPED BY : Mark Arnold
WRITTEN BY : Mark Arnold, Anthony Boyd
TESTED BY : Mark Arnold, Anthony Boyd, Susan Dalton

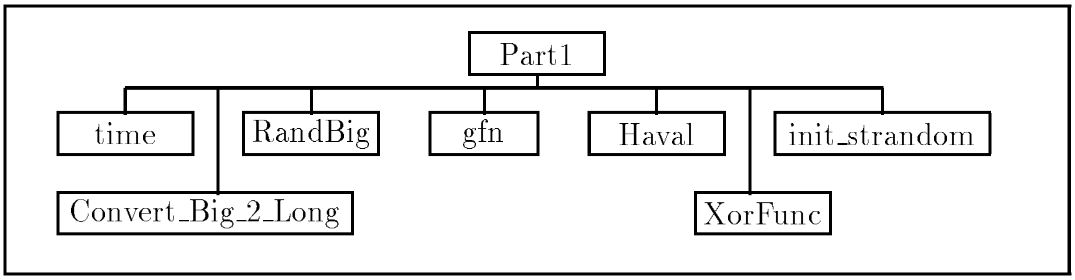

Figure 14: Module Hierarchy of Part 1

FUNCTION PROTOTYPE

```
static void
Part1 (xb,ya,p,g,m)
long *xb;
long *ya;
long *p;
long *g;
char *m;
```

PARAMETERS

yb	: This is the public key of the other user.
xa	: This is the private key of the user.
p	: This is the prime used in the generation of the user's public key.
g	: This is the generating number used in the generation of the user's public key.
m	: This is a pointer to the generated message to be hashed.

VARIABLES

x	: This pointer to a long will point to the random number generated.
xbx	: This variable is used to store the sum of the users private key and the random number x.
one	: This contains the representation of one as defined by the lenstra_2 routines for large numbers.
c1big	: This variable is used for the storage of cipher text 1 in the representation defined by the lenstra big number routines.
upper	: This variable is used as the upper limit for the random number generator.
r	: This is used to store the value of $y_A^{x_B+x}$ mod $prime$.
c1	: This is the representation of c1big as a string of 32-bit longs.
c1_len	: This is the length of the ciphertext $c1$ in words.
cc	: As described previously this integer variable is used to determine the

	status of a write to the other communicating party.
s, a, i	: These are loop variables.
m_length	: This is the length of the message in long integers.
gentime	: This variable is used to store the time which in turn is used as a seed for the random number generator.
mesgpr	: This is the concatenation of the message and the variable r, which is the input to the hash function Haval.
digest	: This the the output of Haval or the compressed message.
mesgpdigest	:This is the concatenation of the message and the digest.
c2	: This is the variable for ciphertext 2.
z	: This is a pointer to the output generated by the g-function which is in turn used as input to the XOR-function.
seed	: This is a 32 bit seed used for random number generation.

ABOUT THE FUNCTION

This routine is an implementation of algorithm 1, as described in this manual. Initially variable is set which indicates the length of the message that was previously generated.

A call is made to determine the time which is used to set the seed within the random number generator 'strandom'. The upper limit for the random number is then determined by calculating $p-1$ and a random big number is then generated.

To calculate ciphertext 1 c_1 the large number g is raised to the power x and the result mod p is calculated by a call to the zexpmod() large number routine, this ciphertext is then converted to a string of long integers for loading into the buffer and transmission to the other party.

To generate ciphertext 2 c_2 the following is done, the random number x is added to the users private key xb, the result of which is used in the following calculation of $r = (yb^(xa + x))$ mod p, the modulus is also obtained by use of the appropriate large number functions. A bit string z is then generated by using r as input to a cryptographically strong pseudo random string generator known as the g-function. The variable r is also appended to the message which is then compressed using the hashing function Haval. The digest is then appended to the original message and this is exclusive-ORed with the generated z to produce c_2.

Now that c1 and c2 have both been generated, they are both copied into a buffer and written to the other party, and the routine returns to its caller.

NAME OF MODULE : RandBig
DESIGNED BY : Mark Arnold
DEVELOPED BY : Mark Arnold
WRITTEN BY : Mark Arnold
TESTED BY : Mark Arnold

FUNCTION PROTOTYPE

void
RandBig(upper,result)
long *upper;
long **result;

PARAMETERS

upper : this is the upper limit for which the generator will produce a number for.
result : the result of the random number generation will be placed here.

VARIABLES

long mask : a mask so that the lower 31 bits can be removed
long lokiout[2] : the output of loki will be placed here
long tmp : this number will be multiplied to produce a new number
boolean finished : a loop control.

ABOUT THE FUNCTION :

This routine iterates through a loop which continually encrypts a number which gets masked for its lower eight bits these are then multiplied with the current number generated to produce a new number. If this new number is greater then the upper limit, then the previous result is the number which is returned otherwise the loop continues to iterate.

This function is written the way it is to produce a random number which is closer to the upper limit as it can be. The number produced is cryptographically strong, and when it is used in the authentication process makes the system more secure.

NAME OF MODULE : RandLong
DESIGNED BY : Anthony Boyd
DEVELOPED BY : Anthony Boyd
WRITTEN BY : Anthony Boyd
TESTED BY : Anthony Boyd

FUNCTION PROTOTYPE

void
RandLong(upper,result)
long upper;
long *result;

PARAMETERS

upper - This is the upper limit for which the generator will produce a

number for.
result - The result of the random number generation will be placed here.

VARIABLES

long mask - A mask so that the lower 16 bits can be removed
unsigned int strandout - The output from the strandom routine will be placed here
unsigned int add - Used as the increment in producing the new number.
boolean finished - a loop control.

ABOUT THE FUNCTION

This routine iterates through a loop which continually generates a number which gets masked for its lower sixteen bits, these are then added to the current number generated to produce a new number. If this new number is less then the upper limit, then the number produced is returned otherwise the loop continues to iterate.

NAME OF MODULE : Big_Long
DESIGNED BY : Susan Dalton.
DEVELOPED BY : Mark Arnold and Susan Dalton.
WRITTEN BY : Mark Arnold and Susan Dalton.
TESTED BY : Mark Arnold and Susan Dalton.

FUNCTION PROTOTYPE

```
int
Convert_Big_2_Long (x, a) long *x;
long *a;
```

PARAMETERS

x - This is a pointer to a bigint structure as defined in the lenstra_2 package for big numbers, which contains the bigint to be converted to a string of long integers.
a - This is a pointer to the start of the returned string of long integers, which has previously been allocated sufficient memory.

VARIABLES

i - This variable is used as a loop index, which also indexes into the elements of the bigint as it is converted.
p - This pointer is used to indicate the current position of the process where the next long integer will be placed in the returned string.

ABOUT THIS PROCEDURE

This routine utilises a local sub-function Conv2L to do most of the work. The call to Conv2L varies depending on the how advanced the process is as the bits needed to be masked out for the next long integer in the returned string changes.

Finally the length of the number of long integers in the string is returned.

FUNCTION PROTOTYPE

```
static void
Conv2L (i, x, big, msk1, msk2, shiftR, shiftL)
int             i;
long            *x;
long            *big;
long            msk1,
                msk2;
short           shiftL,
                shiftR;
```

PARAMETERS

i	- This is the index into the bigint which is being converted into a string of long integers.
x	- This is a pointer to the position where the next element in the string of longs will be placed.
big	- This is a pointer the the bigint currently being converted.
msk1	- This is the first mask used to select the appropriate bits out of the current element in the bigint which is being converted to a string of long integers.
msk2	- This is the second mask used to select the appropriate bits out of the next element in the bigint which is being converted to a string of longs.
shiftL	- This is the amount of bits the current element in the bigint needs to be shifted left before being masked and incorporated into the string of long integers.
shiftR	- This is the amount of bits the current element in the bigint needs to be shifted right before being masked and incorporated into the string of long integers.

VARIABLES

tmp1	- This is a long integer used when processing the current element of the bigint.
tmp2	- This is a long integer used when processing the next component of the bigint.
result	- This is the resultant element which is appended to the string.

ABOUT THE FUNCTION

Initially the variables tmp1, and tmp2 are set to the current and next elements of the bigint as determined by the index i. These values are then masked so as to assure that they are in the correct format clearing any erroneous bits. These variables are then shifted the appropriate number of bits to the right and left as determined by the passed shift counts. The required bits are then masked out and bitwise added together finally being appended onto the string.

NAME OF MODULE : gfn
DESIGNED BY : Mark Arnold
DEVELOPED BY : Mark Arnold
WRITTEN BY : Mark Arnold
TESTED BY : Mark Arnold

FUNCTION PROTOTYPE

```
long*
gfn(num, length)
long *num;
int length;
```

PARAMETERS

num : This is a pointer to a long which is the input to the g-function.
length : This variable indicates the number of long integers the caller would like in the output.

VARIABLES

i	: loop variable
key	: This array of 2 long integers is used as a 64 bit key for loki91.
input	: This array of 2 long integers is used as the input to loki which is encrypted, to obtain a random result.
retptr	: This pointer to a long is what is returned, the return value being a random string of num long integers in length.

ABOUT THE PROCEDURE

The first step in this routine is to initialise the input to be encrypted by loki, and thus the input array is set to zero, also size is allocated for result which is to be returned. The key for loki91 is set to the input and the process of generating a random bit string begins. The routine loops taking 64 bits at a time from a fixed encrypted input and adjusting the key as necessary. If the length is odd then a final call to the loki routines are made to generate the last 32 bits, finally a pointer to the random generated string is returned.

A message is compressed to produce a digest by calling the Haval function. Haval then calls various other functions to perform the message compression to create the digest.

NAME OF MODULE : Haval
DESIGNED BY: Y. Zheng, J. Seberry, J. Pieprzyk
DEVELOPED BY : Y. Zheng
WRITTEN BY : Y. Zheng
TESTED BY : Y.Zheng

PARAMETERS

short
haval_string(string, fingerprint)
char *string
unsigned char *fingerprint

PARAMETERS

string	- This is the string to be hashed
fingerprint	- This is the output of the hash function or the compressed message.

VARIABLES

haval_state	- This holds current state information being the number of bits in the message, the current state of the fingerprint, a block of input characters and any residual characters or remaining input
len	- This is used to store the length in characters of the input string.

ABOUT THE PROCEDURE

Haval starts by initialising the fingerprint and then calls the function haval_hash() to do most of the work. This is followed by a call to the routine haval_end() to adjust the size of the digest if necessary.

NAME OF MODULE: XORFunc
DESIGNED BY: Anthony Boyd
DEVELOPED BY: Anthony Boyd
WRITTEN BY: Anthony Boyd
TESTED BY: Anthony Boyd

FUNCTION PROTOTYPE

unsigned char*

```
XORFunc(mesg1, mesg2, length)
unsigned char* mesg1;
unsigned char* mesg2;
int length;
```

PARAMETERS

The XORFunc takes three parameters and returns a pointer. The three parameters in order are mesg1 which is a pointer to an unsigned character, mesg2 another pointer to an unsigned character, and finally length which is simply an integer.

These parameters represent the two arrays which need to be exclusive-ORed and the length of each array. The return value from this function is the array which is the result of this operation.

VARIABLES

i	- This is used simply as a loop control variable.
retptr	- This the the pointer which is returned to the calling function This pointer will point to the result of the exclusive-OR.

ABOUT THIS FUNCTION

The function systematically exclusive-ORs each element in the two arrays of unsigned characters passed in as parameters and stores the result in another array of unsigned characters, upon termination returning a pointer to this array. By doing this it makes it impossible to determine what the original arrays contained without having one of them to exclusive-OR the resultant array. This makes whatever the original array contained extremely secure. This function is called in the authentication process.

3.3.3 Part 2

This section of code performs the second part of the authentication process, by implementing algorithm 2 of the authentication algorithms used in this system (see section 2.4). Part 2 module hierarchy is shown in figure 3.3.3.

NAME OF MODULE : Part2
DESIGNED BY : Y. Zheng, J. Seberry
DEVELOPED BY : Mark Arnold
WRITTEN BY : Mark Arnold, Anthony Boyd
TESTED BY : Mark Arnold, Anthony Boyd, Susan Dalton

FUNCTION PROTOTYPE

```
static boolean
```

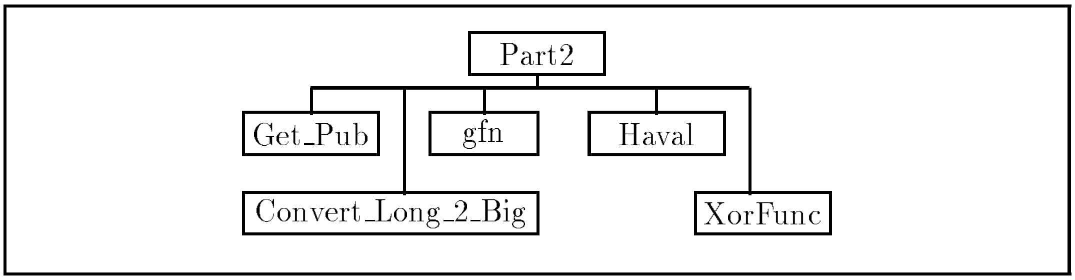

Figure 15: Module Hierarchy for Part 2

```
Part2 (xa,yb,p,c1,c2)
long *xa;
long *yb;
long *p;
c1_type c1;
c2_type c2;
```

PARAMETERS

xa	: This is the private key of the user.
yb	: This is the public key of the other person.
p	: This is the prime used in the generation of the users private and public key pair.
c1	: This is a structure containing ciphertext 1 generated in the first stage of authentication
c2	: This is a structure containing ciphertext 2 generated in the first stage of authentication

VARIABLES

ybc1	: This pointer to a long will point to product of ciphertext 1 and the users public key.
r	: This contains $yb.c1^{xa}$ mod p (where p is the prime used in generation of the users private and public keys).
cipher1	: This variable is used to hold a large number representation of the ciphertext 1 (c_1).
m	: This is a pointer to the original plaintext message m as generated in the first part of the authentication process.
mesgpr	: This is a pointer to the concatenation of the message m and the variable r.
m1	: This is a pointer to the message which is restored from the XOR function, to its original contents (providing it has not been tampered with).
z1	: This is a pointer to the output generated by the g-function which is in turn used as input to the XOR-function.

first_digest	: This is for the digest which was calculated by the other party in the first stage of authentication.
second_digest	: This is for the digest we calculate for comparison against the other parties calculated digest.
fptlen	: This is the length of the precomputed digest.
i	: These is a loop variable.

ABOUT THE FUNCTION

This routine is an implementation of algorithm 2, as described in this manual. Initially space is set for cipher1 to be stored as a large number, cipher1 is then converted to a large number by a call to the appropriate routine. The first step in the algorithm is to take cipher1 multiply it by the other persons public key, the result of which is raised to our private key, to produce $(yb.c1)^{xa}$ we then take the result modulo the prime that was passed into this procedure. This value is stored in the variable r. This is used as the input for the g-function to generate z', from which $m1$ the original message obtained by exclusive-OR. The variable r is then appended to the original message and this is compressed using the hashing function Haval. The precomputed tag or digest is then compared with the new digest, and if equal authentication has been successful otherwise it has failed.

NAME OF MODULE : Long_Big
DESIGNED BY : Mark Arnold
DEVELOPED BY : Mark Arnold and Susan Dalton
WRITTEN BY : Mark Arnold and Susan Dalton
TESTED BY : Mark Arnold and Susan Dalton

FUNCTION PROTOTYPE

```
void
Convert_Long_2_Big (p, big, length)
long *p;
long *big;
int length;
```

PARAMETERS

p	: This is a pointer to a string of long integers to be converted into a bigint structure as defined in the lenstra_2 package for large numbers.
big	: This is a pointer to the start of the returned bigint which has proviously been allocated sufficient memory.
length	: This is the length in the number of long integers to be converted, into a bigint structure.

VARIABLES

i : This variable is used as a loop index, which also indexes into the elements of the bigint as it is created from the string.

ABOUT THIS PROCEDURE

This routine utilises a local sub-function Conv2B to do most of the work. The call to Conv2B varies depending on the how advanced the process is, as the bits needed to be masked out for the next element in the bigint returned changes.

FUNCTION PROTOTYPE

```
static void
Conv2B (i,p,big, msk1, msk2, shiftR, shiftL)
int i;
long *p;
long *big;
long msk1,
msk2;
short shiftR,
shiftL;
```

PARAMETERS

i : This is the index into the bigint which is being created from the string of long integers. p : This is a pointer to the position where the next element in the string of longs will be obtained for use in the conversion process. big : This is a pointer the the bigint currently being created msk1 : This is the first mask used to select the appropriate bits out of the current element in the string of long integers. msk2 : This is the second mask used to select the appropriate bits out of the next element in the string of long integers. shiftL : This is the amount of bits the current element in the string needs to be shifted left before being masked and incorporated into the bigint. shiftR : This is the amount of bits the current element in the string needs to be shifted right before being masked and incorporated into the bigint.

VARIABLES

ptmp : This is a copy of the current element in the string which is manipulated, and will be incorporated into the current element in the bigint. ptmp1 : This is also copy of the current element in the string which is manipulated, and will be incorporated into the next element of the bigint.

ABOUT THE FUNCTION

Initially the copies of the current element are shifted left and right by the appropriate shift counts as passed to the function, following this the appropriate bits are masked off by use of the appropriate masks as passed to the function. The current and next elements are then created by the use of bitwise addition, finally the two elements of the bigint are masked so as to be in the correct format.

NAME OF MODULE : GetPub
DESIGNED BY : Mark Arnold and Adam Millard
DEVELOPED BY : Mark Arnold and Adam Millard.

WRITTEN BY : Mark Arnold.
TESTED BY :

FUNCTION PROTOTYPE

```
int
GetPub (publickey, g, prime, name)
long **publickey;
long **g;
long **prime;
char *name;
```

PARAMETERS

publickey	- This is the users public key.
g	- This is the random number used in the creation of the users public key.
prime	- This is the prime number used in the creation of the users public key.
name	- This is the other users login name.

VARIABLES

pubkey_file	- This is a pointer to the users public key file.
l	- This is a pointer to the full name (including the directory path) of the users public key file.

ABOUT THE FUNCTION

The name for the public key file is created using the login name passed to the function in the parameter 'name'. This is done by combining the users login name with a prefix (which is the path to the directory in which the users keys are to be found) and an extension (in this case '.pub').

Then the public key file for the other user is opened and their public key, g, and prime are read from it, and the memory allocated to the other users public key file name is deallocated.

This function returns a value of '1' to the calling program if the file is successfully opened, otherwise returning a value of '-1'.

3.4 Security

These functions perform the encryption and decryption of all message when a user selects the security option.

NAME OF MODULE : encryptbuf

DESIGNED BY : Mark Arnold
DEVELOPED BY : Mark Arnold and Anthony Boyd
WRITTEN BY : Anthony Boyd and Mark Arnold
TESTED BY : Anthony Boyd and Mark Arnold

FUNCTION PROTOTYPE

```
void encryptbuf(encbuf, num)
char encbuf[bufsize];
int num;
```

PARAMETERS

char encbuf[bufsize] : This is the actual buffer which will be encrypted and then sent across to the other user.
int num : This is the number of characters in buf which need to be encrypted.

VARIABLES

int i : a loop control variable.

ABOUT THE FUNCTION

To encrypt the buffer for sending across the communication channel, several things need to be done. Initially the buffer is encrypted by exclusive-OR with the random global random bitstring 'enc'. After all the characters have been encrypted a check is made to see if a new 'enc' needs to be generated, and if necessary a new bitstring is created.

NAME OF MODULE : decryptbuf
DESIGNED BY : Mark Arnold
DEVELOPED BY : Mark Arnold and Anthony Boyd
WRITTEN BY : Anthony Boyd and Mark Arnold
TESTED BY : Anthony Boyd and Mark Arnold

FUNCTION PROTOTYPE

```
void decryptbuf(decbuf, num)
char decbuf[bufsize];
int num;
```

PARAMETERS

char decbuf[bufsize] : This is the actual buffer which will be decrypted and then displayed to the user.

int num : This is the number of characters in buf which need to be decrypted.

VARIABLES

int i : a loop control variable.

ABOUT THE FUNCTION

This function is essentially the same as encryptbuf(). To decrypt the buffer for displaying the plaintext for the user several things need to be done. The buffer contents are initially exclusive-ORed with the bitstring dec to decrypt the buffer. Then a check is made to see if a new bitstring is need and if so appropriately created.

NAME OF MODULE : generatekey
DESIGNED BY : Mark Arnold
DEVELOPED BY : Anthony Boyd and Mark Arnold
WRITTEN BY : Anthony Boyd
TESTED BY : Anthony Boyd

FUNCTION PROTOTYPE

```
void generatekey(key1,key2)
unsigned int key1[2];
unsigned int key2[2];
```

PARAMETERS

unsigned int key1[2] : This is the first part of the classical key which the user created themselves and sent to the other user as part of the authentication message.
unsigned int key2[2] : This is the second part of the classical key which the user extracted from the authentication message sent by the other user.

VARIABLES

int t : a loop control.

ABOUT THE FUNCTION

This function simply takes in the two minor keys and exclusive-ORs them to produce the actual classical key used in the security process. This classical key is then set by a call to init_strandom().

3.5 Code Modified in 'Talk'

3.5.1 io.c

This file was modified to contain two different versions of the 'talk' procedure within the original 'talk' procedure. This was done to differentiate between what was required during a 'normal' session of talk of what was required during a "secure" session of talk. A 'secure' session of talk entails encrypting and decrypting the buffer of characters sent between the two users.

In the 'normal' talk function no changes were made to the original talk code. However in the 'secure' talk part of the function calls were made to encrypt and decrypt the buffer.

3.5.2 talk.c

This file was modified to contain a call to the authentication function if authentication was required.

3.5.3 get_names.c

This file was modified to include conditions for the reading of the tty from the command line since there will also be options entered on it. These conditions were necessary since the application might mistakenly read a option from the command line and treat it as a tty.

3.5.4 init_disp.c

This file was modified to include Secure Talk specific entries in the initial buffer which contains the users editing characters. Specifically the function set_edit_chars was changed to include a larger buffer to accommodate the extra options which are sent across. These options determine what mode of operation Secure Talk is operating in. ie. the authenticate and security flags are placed in this buffer. This function also determines if duel authentication will be taking place.

3.6 Changes Made To Lenstra_2

The function szread was added to the lenstra_2 package for large numbers, this new routine is passed a pointer to a character string which will be read in as a bigint structure, the routine is also passed the address of a bigint structure in which the result is returned. The szread routine repeatedly loop for each character in the (null terminated) string building the large number up digit by digit using the routines zsmul (string multiplication routine) and zsadd (string addition routine).

References

[SZ93] J. Seberry and X.-M. Zhang. Highly nonlinear 0-1 balanced boolean functions satisfying strict avalanche criterion, 1993.

[Zhe93] Y. Zheng. STRANDOM — a cryptographically strong pseudo-random number generator based on HAVAL, July 1993. (source code in the programming language C).

[Zhe94] Y. Zheng. Improved public key cryptosystems secure against chosen ciphertext attacks, February 1994.

[ZPS93] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL — a one-way hashing algorithm with variable length of output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology - Proceedings of AusCrypto'92,* Lecture Notes in Computer Science, Vol. 718, pages 83–104. Springer-Verlag, 1993.