ADMINISTRATOR'S GUIDE TOTHE DIGITAL SIGNATURE FACILITY "ROVER"

Matt Bishop

Technical Report PCS-TR90-153

August 1990

Administrator's Guide to the Digital Signature Facility "Rover"

Matt Bishop[1]

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

ABSTRACT

This document describes the installation and maintenance of the *rover* utility, which provides a digital signature capability for internet messages.

1. Introduction

This document contains installation instructiona and examples of use of the *rover* facility. This facility is not a general key management facility, nor is it intended to provide authentication of users; assuming the system is installed and maintained correctly, as described below, it simply guarantees that a message purporting to originate from a specific user did in fact come from that user (or someone who possesses that user's cryptographic key). The mechanism used is described in [1]; for a more detailed description of how this program works, see the associated document [2].

In what follows, file names in **boldface** are real file names; file names in *italics* should be replaced by the relevant file names on your system. Smetimes shell variables are relevant; these are also indicated by **boldface**. Variables defined in the relevant makefile use the syntax of a makefile variable reference; for example, ${*makefile_variable*}. Finally, specific host names are in **boldface** and a name that is to be replaced by a host name will be in *italics*.

2. Configuring and Compiling the *rover* Libraries and Server

This package can be compiled on either Berkeley UNIX[2] or System V UNIX computers with no changes. Other versions of UNIX may require some changes.

1. Determine whether your system is closer to System V or Berkeley UNIX. Type

   sh ~~Install~~ setup.sh

   and answer "bsd4" or "sysv" when prompted. This will set up the appropriate **Makefiles**.
2. Edit **Makefile** and the **Makefiles** in the subdirectories **rover**, **net**, and **seal**. The parameters which may have to be reset are described in section 4.
3. Switch to the superuser and compile and install the software:

   make install
4. Register your users; see section 6.
5. Go home! You're all done.

---

1. Work done at the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA 94035 and supported by award NCC 2-397 from the National Aeronautics and Space Administration to the Universities Space Research Association.
2. UNIX is a Registered trademark of AT&T Bell Laboratories.

3. Source Organization

The source to *rover* is organized in several different directories:

include/	contains include files peculiar to *rover*; this does *not* contain those include files for the libraries **net** or **seal**
net/	contains source for the network library used by *rover*; this library provides a simple interface to the Berkeley UNIX TCP/IP interface
rover/	contains the source for the *rover* server and the database manager
seal/	contains the cryptographic signature and validation routines

These each have a makefile, and **net**, and **seal** will compile into separate libraries which may be used by programs other than *rover*. Those three directories also contain test programs.

Note that the makefiles in the subdirectories are tailored for use on the developmental system; this means that they may, or may not, work on your system. This is usually irrelevant, because the master makefile, **Makefile**, in the top-level directory passes the appropriate parameters to the lower-level make in such a way as to override the settings in the makefiles in the subdirectories. If those makefiles are to be modified so they can be used in the subdirectories, the parameters should be changed as in the following section.

4. Makefile

There are two types of Makefiles: the one in the source root directory, which just calls the others, and the one in each of the **rover**, **net**, and **seal** subdirectories. This section describes variables found in all of them; edit the top-level one first, then each of the ones in the subdirectories as necessary. The **Makefiles** contain several variables that can be changed to compile and install the software properly on your system. This section summarizes those variables.

4.1. Makefile Programs and Environment

The make executes a number of programs; in the course of setting up the relevant dependencies. The following makefile variables should be set appropriately for your system:

variable	*sample value*	*what it means*
BASENAME	basename	the command *basename*(1)
MAKE	make	the command *make*(1)
RM	rm -f	the command *rm*(1), here with an option to force removal
SHELL	/bin/sh	the Bourne shell *sh*(1)

Note that the setting of the value of ${*SHELL*} is critical on System V-based computers, which use the value of that variable as the shell to execute the commands under the dependencies. In particular, the commands are designed to be run under the Bourne shell, not *csh*(1), so if the ${*SHELL*} variable is set incorrectly, the libraries and executables will not build properly.

4.2. Installation Parameters

These parameters control the installation of the libraries and programs. The following makefile variables should be set appropriately for your system:

variable	*sample value*	*what it means*
USER	bin	the owner of the libraries and executables
GROUP	staff	the group of the libraries and executables
LIBDIR	/usr/local/lib	the directory into which the libraries are to be copied
ROVERDIR	/usr/local/etc	the directory into which the *rover* server is to be placed
INSTOPT	-c -g $(GROUP) -u $(USER)	the options to *install*(1)

4.3. Library Construction Parameters

The make executes a number of programs in the course of compiling and building libraries. The following makefile variables should be set appropriately for your system:

variable	*sample value*	*what it means*
AR	ar rcv	the command *ar*(1), with options to create a new library
LINT	lint	the command *lint*(1)
LORDER	lorder	the command *lorder*(1)
RANLIB	ranlib	the command *ranlib*(1)
TSORT	tsort	the command *tsort*(1)

The makefiles contain these as commands with file names as arguments, so if any of these are not present on your system (for example, System V-based UNIXes often do not have *ranlib*), set them to *true*(1). This program simply exits, returning success.

4.4. Compilation and Syntax Checking Parameters

The make compiles a number of programs and libraries. The following makefile variables should be set appropriately for your system:

variable	*sample value*	*what it means*
COPTS	-g	the flags passed to *cc*(1), except for -D... and -I...
DEFS	-DBSD4 -DDES	predefined macros (see below)
INCS	-I../include	directories with header files
LOPTS	-uphbac	the flags passed to *cc*(1), except for -D... and -I...
LOUT	-C$(LLIB)	the flag passed to *lint*(1) for generating the lint library

The makefiles pass these as arguments to the compiler and syntax checker. Note that defined macros should be set in *DEFS*, and include paths in *INCS*, not in *COPTS*. In addition to the usual ones (see *cc*(1)), the following predefined macros are useful:

-DBSD4	Set this if your version of UNIX is (or is derived from) the Fourth Berkeley Software Distribution.
-DSYSV	Set this if your version of UNIX is (or is derived from) System V.
-DCRAY	Set this if you are using a Cray running UNICOS 5.0 or later.
-DINETD	Set this if you are running the network daemon initializer *inetd*(8); if this is not set, *rover*'s start up routine will simulate the *inetd* environment and then invoke *rover*.
-DDEFLOGFILE	Set this to the name of the file into which *rover* is to log information. The default value is in **rover/rover.h**.
-DROVERHOST	Set this to the name of the host on which the *rover* server sits. It can be in any form that can be mapped to an internet address (so if it's local, you probably don't need the fully qualified domain name).
-DROVERPORT	Set this to the port number on which the *rover* server is to listen.
-DCAESAR	Set this to have the encryption and integrity protection done using a Cæsar cipher. This is not recommended in practise.
-DDES	Set this to have the encryption and integrity protection done using the Data Encryption Standard cipher in cipher block chaining mode [3][4].

5. Testing the Libraries and *rover*

This section describes how to test each of the libraries separately, to be sure they function, and then how to test *rover*. The order of testing is important, as if something is not working, nothing following its section will work either. So we suggest you follow the order of these sections.

5.1. libseal.a

This library does the cryptographic signing, and encryption (when requested). All cryptographic routines are isolated in the file **c_funcs.c**; if you want to add a new cryptosystem, do so there. (The file header contains the interface specifications.)

The library test involves two programs, *tests.c* and *testu.c*. The first of these simply seals its input and writes it to the output; the second unseals whatever it gets from the input and writes it to the output. Manual pages are included for both.

The following steps show how the library can be tested.

1. Make the library and the executables by modifying the local **Makefile** appropriately and typing

   make all

2. Issue the command

   tests tests.c *sealedfile*

   This command will cryptographically seal the contents of the file **tests.c** and write it to *sealedfile*. If you wish, you may have the contents encrypted as well by giving the "-e" option to **tests**. (You can also use an origin and a destination other than you; see the manual page *tests*(1) for appropriate options.)

3. Issue the command

   testu *sealedfile unsealedfile*

   This command will unseal the contents of the file *sealedfile* and write it to *unsealedfile*. If you want to verify the correct origin and destination, give the "-v" option; this will print the origin, destination, and size of each sealed packet.

4. Compare the contents of **tests.c** and *unsealedfile*:

   diff **tests.c** *unsealedfile*

   They should be identical.

5.2. liblnet.a

This library provides a (much) simplified interface to the Berkeley TCP/IP interface.

The library test involves two programs, *tests.c* and *testc.c*. The first of these is a server which reads lines of text from clients, prepends a ">", and writes the result back to the client; the second is a client which connects to the server, transmits a file, and prints whatever the server sends back. Manual pages are included for both.

The following steps show how the library can be tested.

1. Make the library and the executables by modifying the local **Makefile** appropriately and typing

   make all

2. Issue the command

   tests -l 'pwd'/logfile &

   This command will start the server in the background. By default, the server will log connections to **logfile** in the root directory, and will listen for connections on port number 6789. (You can also use a different log file and port number; see the manual page *tests*(1) for appropriate options.) Do not be alarmed if the process appears to exit immediately; the server spawns a child which does the actual work. (On some systems, a grandchild may do the actual work to avoid a controlling terminal ever being assigned; see the routine *inetsetup()* in **inetinit.c** if you're really curious.)

3. Issue the command

   testc *file arrowfile*

   This command will read *file*, prepend an ">" to each line, and write it to *arrowfile*. (You can also use a different port number and host; see the manual page *tests*(1) for appropriate options.)

4. Compare the contents of *arrowfile* and the file obtained by prepending ">" directly:

   sed 's/^/>/' *file* | diff - *arrowfile*

   They should be identical.

5.3. Testing *rover*

Testing *rover* is rather straightforward once the libraries are tested. Essentially, you build and install the server, and then use the dummy database to seal and unseal a message.

The cryptographic keys are kept in a database which can be edited by the program *dbm*.

Currently, two keys are defined: the first, for seal@local, is "testin", and the second, for unseal@-remote, is "testout". The program *tseal* will act as though "seal@local" were sending something to "unseal@remote" during this test.

The following steps show how this can be tested.

1. Make *rover*, the database editor *dbm*, and the test programs by modifying the local **Makefile** appropriately and typing

   make all

2. Next, build the cryptographic database to be used for testing:

   dbm -s test.input -q

   This will set up the database so that *rover* can be tested. **Important: if test.dbm exists, delete it first** as it is a *binary* database, and may not be correctly interpreted by your computer.

3. Issue the command

   rover -l 'pwd'/test.log -r 'pwd'/test.dbm &

   This command will start the *rover* server in the background. By default, the server will log connections to **rover.log** in the root directory, and will obtain cryptographic keys from **rover.dbm** in the root directory. The two options reset the log and key file names to be **test.log** and **test.dbm** in the current directory.

3. Issue the command

   tseal -lseal -Llocal -runseal -Rremote *file savefile*

This command will read *file*, split it up into messages, cryptographically seal each, and put the result in *savefile*. When prompted for the *rover* key, type:

testin

4. Issue the command

   tunseal *savefile newsavefile*

This command will read *savefile*, cryptographically unseal each of its messages, and put their concatenation in *newsavefile*. When prompted for the *rover* key, type:

testout

4. Compare the contents of *file* and *newsavefile*:

   diff *file newsavefile*

   They should be identical.

6. Administering the *rover* Database

The heart of *rover* is a database associating *users* with *cryptographic keys*. This file is very sensitive and should always be kept on a protected machine; if it is compromised, the whole *rover* mechanism is undependable. **It is recommended that the *rover* server, and this database, be kept on a physically protected computer on which the only allowed network activity are connections to the *rover* server. To log in as a user must require physical presence in the control room, where the user can be observed. Without this protection, *rover* will not provide the necessary assurance of authenticity.**

The program to manage the database is *dbm*; it is described in the manual page. To enter users into this database, run *dbm* and add users with the a command. For example, to enter the users "seal" and "unseal" used in the previous section, issue the following commands (the computer's responses are in **boldface**, and comments are in *italics*):

`dbm -r test.dbm`		*invoke dbm on the database "test.dbm"*	
`> a seal local testin`			
		add the user "seal" on host "local" with key "testin"	
`> a unseal remote testout`			
		add the user "unseal" on host "remote" with key "testout"	
`> p`		*print the database contents*	
record #	**status**	**who**	**value**
0	**active**	**<seal@local>**	**<testin>**
1	**active**	**<unseal@remote>**	**<testout>**
`> q`		*quit, saving the contents of the database*	

The commands in the previous section did the same thing, but using a file that was read from the command line. However, note that when this is done, when *dbm* has finished reading the file, it returns to command level; the extra -q causes it to exit.

7. Adding a New Cryptosystem to *rover*

To add a new cryptosystem, you have to modify four functions in the file **seal/c_funcs.c**:

c_pwsize()
: returns the length of the longest acceptable cryptographic key;

c_mic(keylen, key, begin, end)
: computes a message integrity check using the cryptographic key **key** of length **keylen**; the buffer to be checked begins at **begin**, with **end** pointing to the address just *beyond* the end of the buffer. It is expected to return a pointer to a set of ASCII characters representing the integrity check. These characters may be in a static array that is overwritten with each call.

c_encrypt(keylen, key, begin, end)
: uses the cryptographic key **key** of length **keylen** to encrypt the buffer beginning at **begin**, with **end** pointing to the address just *beyond* the end of the buffer. The encryption is to be done in place. It is guaranteed that the buffer's length is a multiple of 8 bytes.

c_decrypt(keylen, key, begin, end)
: uses the cryptographic key **key** of length **keylen** to decrypt the buffer beginning at **begin**, with **end** pointing to the address just *beyond* the end of the buffer. The decryption is to be done in place. It is guaranteed that the buffer's length is a multiple of 8 bytes.

You must write these routines for your cryptosystem. Note that if you have cryptographic apparatus (hardware) to hold the keys, you should alter the functions in **seal/crypto.c** to take advantage of it; that way, the keys need never appear in memory.

8. References

[1] R. Merkle, "Protocols for Public-Key Cryptosystems," *Proceedings of the 1980 Symposium on Privacy and Security* (Apr. 1980) pp. 122-133.

[2] M. Bishop, "A Digital Signature Mechanism," Technical Report PCS-TR90-154, Dartmouth College, Hanover, NH (*in preparation*).

[3] *Data Encryption Standard*, FIPS PUB 46, Department of Commerce, Washington, DC (1976).

[4] *DES Modes of Operation*, FIPS PUB 81, Department of Commerce, Washington, DC (1978).

MANUAL PAGES

NAME

dbm – rover database management and editing program

SYNOPSIS

dbm [*commands*]

DESCRIPTION

Dbm manipulates a database of user information and cryptographic keys used by the digital signature system *rover*. The database contains sets of triplets consisting of user name, host name, and key (see *dbm*(5) for the exact format). Available commands are:

a *name host key*
: Add the triplet (*name*,*host*,*key*) to the database; this command creates an entry saying that *name*@*host*'s key is *key*.

d *name host*
: Delete the etry for *name*@*host*.

f*name host*
: Fetch the key associated with *name*@*host*.

h, ? Print a help message.

i Print information about the database (name of the database file and the number of active and deleted records).

p Print the contents of the database.

q Quit, saving all modifications to the database.

r *file* Read the contents of *file* using them as the database. This closes any current database, and makes *file* the new one. **s** *file* Read commands to *dbm* from *file*. Note that giving a **q** in the file returns you to interactive mode rather than terminate the editing session.

t *file* This saves the contents of the database in a portable format, essentially creating a set of command lines that when given to *dbm* using the **s** command will reconstruct the database. As database files are not in general portable (they contain some binary data), this mechanism can be used to transfer databases between systems with different architectures.

Comment; ignore this line.

! *command*
: execute *command* by passing it to a subshell.

OPTIONS

Any command can be given as an option; command-line arguments are executed first, then if neceaasry the user will be prompted for input.

SEE ALSO

rover(1), dbm(5)

NAME

rover – digital signature server

SYNOPSIS

rover [**–l***logfile*] [**–r***database*]

DESCRIPTION

Rover is a digital signature server. It accepts connections, reads messages from one user to another cryptographically sealed using the originator's key. It then validates the message and the originator, and reseals the message using the destination's key; the destination process can then unseal the message and read it. The server vouches for the authenticity of the claimed origin and for the integrity of the message,

OPTIONS

–l*logfile*

Log to the file instead of the default "/etc/rover.log".

–r*database*

Cryptographic keys are stored in the database *database* instead of the default "/etc/rover.dbm".

FILES /etc/rover.log default log file
/etc/rover.dbm default database file

SEE ALSO

dbm(1)

NAME

testc, tests – test the simple network library

SYNOPSIS

tests [**−l***logfile*] [**−p***portno*]

testc [**−p***portno*] [**−s***serverhost*] [*infile*] [*outfile*]

DESCRIPTION

Tests is a server which accepts connections from clients, reads lines from them, prepends a ">" characxter to each, and writes them back. It provides a demonstration of the use of the routines in the library *lnet*(3).

The server logs all connections into a the log file "test.log" unless the **−l** option is given, in which case logging is done to the file *logfile*. The server accepts connections on port 6789 unless the **−p** option is given, in which case port number *portno* is used.

Testc is a client which talks to *tests*. It assumes the server is listening on port number 6789 unless the **−p** option is given (in which case it uses port number *portno*) and is running on the local host unless the **−s** option is given (in which case it uses the host named *serverhost*). If the input file is not specified or is given as "–", lines are read from the standard input; if the output file is not specified or is given as "–",

SEE ALSO

lnet(3)

NAME

tests, testu — test the sealer and unsealer

SYNOPSIS

tests [**−e**] [**−l***locuser*] [**−L***lochost*] [**−p**] [**−r***remuser*] [**−R***remhost*]] *infile outfile*

testu [**−v**] [*infile outfile*

DESCRIPTION

Tests cryptographically seals a file using the *seal*(3) and *unseal*(3) routines. The file is broken up into a set of messages, and each message is sealed.

Testu takes the output of *tests* and unseals it, generating the input to *tests*.

OPTIONS

−e Encrypt the messages as well as sealing them. The encryption is done and the message is then sealed.

−l*locuser* This option makes the originating user *locuser* instead of the default *from*.

−L*lochost* This option makes the originating host *lochost* instead of the default *from_host*.

−p This sets the **F_PER_MESSAGE** flag in the message headers. It is essentially a no-op and is useful only for debugging. The encryption is done and the message is then sealed.

−r*remuser* This option makes the destination user *remuser* instead of the default *to*.

−R*remhost* This option makes the destination host *remhost* instead of the default *to_host*.

The cryptographic key is obtained by checking for a series of files, and if none are present, prompting at the controlling terminal. Let $*HOME* be the user's home directory and "<SP>" the space character (octal 040, hex 0x20). Then *tests* checks for the files "$*HOME*/..word.tests.*host*<SP>", "$*HOME*/..word.tests<SP>", and "$*HOME*/..word<SP>" in that order. If any exists, is owned by the real UID of the process, and is readable by the owner only, its contents are used as the password. If none of those files meet the criteria, a prompt for the password is sent to the controlling terminal. Similarly, *testu* checks for the files "$*HOME*/..word.testu.*host*<SP>", "$*HOME*/..word.testu<SP>", and "$*HOME*/..word<SP>" in that order. If any exists, is owned by the real UID of the process, and is readable by the owner only, its contents are used as the password. If none of those files meet the criteria, a prompt for the password is sent to the controlling terminal.

SEE ALSO

seal(3)

NAME

tseal, tunseal – test rover

SYNOPSIS

tseal [**–e**] [**–l***locuser*] [**–L***lochost*] [**–p**] [**–r***remuser*] [**–R***remhost*] [**–S***servhost*] [*infile*] [*outfile*]

tunseal [**–v**] [*infile*] [*outfile*]

DESCRIPTION

Tseal cryptographically seals a file using *rover*(1). The file is broken up into a set of messages, and each message is sealed.

Tunseal takes the output of *tseal* and unseals it, generating the input to *tseal*.

OPTIONS

–e Encrypt the messages as well as sealing them. The encryption is done and the message is then sealed.

–l*locuser*
This option makes the originating user *locuser* instead of the default *from*.

–L*lochost*
This option makes the originating host *lochost* instead of the default *from_host*.

–p This sets the **F_PER_MESSAGE** flag in the message headers. A new connection is made to the *rover* server for each packet. The encryption is done and the message is then sealed.

–r*remuser*
This option makes the destination user *remuser* instead of the default *to*.

–R*remhost*
This option makes the destination host *remhost* instead of the default *to_host*.

–S*servhost*
Connect to the *rover* server on host *servhost*.

The cryptographic key is obtained by checking for a series of files, and if none are present, prompting at the controlling terminal. Let $*HOME* be the user's home directory and "<SP>" the space character (octal 040, hex 0x20). Then *tseal* checks for the files
$*HOME*/..word.tseal.*host*<SP>
$*HOME*/..word.tseal<SP>
$*HOME*/..word<SP>
in that order. If any exists, is owned by the real UID of the process, and is readable by the owner only, its contents are used as the password. If none of those files meet the criteria, a prompt for the password is sent to the controlling terminal. Similarly, *tunseal* "$*HOME*/..word.tunseal.*host*<SP>",
$*HOME*/..word.tunseal.*host*<SP>
$*HOME*/..word.tunseal<SP>
$*HOME*/..word<SP>
in that order. If any exists, is owned by the real UID of the process, and is readable by the owner only, its contents are used as the password. If none of those files meet the criteria, a prompt for the password is sent to the controlling terminal.

SEE ALSO

rover(1), seal(3)

NAME

cmphost – see if two host names belong to the same host

SYNOPSIS

#include net.h

int cmphost(host1, host2)
char *host1, *host2;

DESCRIPTION

The function *cmphost* takes two host names as arguments; these may be official names, aliases, or the Internet numbers of the hosts. This function then determines if they represent the same host.

RETURN VALUE

If the two host names represent the same host, 1 is returned; if not, 0 is returned. On error, −1 is returned and *ne_errno* and *ne_call* are set appropriately.

WARNINGS

If the host information is not up to date, the answers returned could be wrong.

Because the system library host information calls return a pointer to a static area, some memory allocation is necessary. The allocation is done using *malloc*(3) and the space is deallocated before return using *free*(3).

SEE ALSO

netperror(3)

NAME

getfdhost – return host at other end of socket

SYNOPSIS

```
#include net.h

int getfdhost(fd)
int fd;
```

DESCRIPTION

The function *getfdhost* takes a file descriptor returned from *netacp*(3) or *netconn (3)* as an argument, and returns the official name of the host at the other end.

RETURN VALUE

On success, a pointer to the official name of the host at the other end of the connection is returned. On failure or error, NULL is returned and *ne_errno* and *ne_call* are set appropriately.

WARNINGS

This routine assumes the connection is done within the Internet domain. If this is not correct, the function will return incorrect information.

If the host information is not up to date, the answer returned could be wrong.

SEE ALSO

netperror(3)

NAME

inetsetup – create a server interface like inted

SYNOPSIS

#include net.h

void inetsetup(portno, func)
int portno;
void (*func)();

DESCRIPTION

The function *inetsetup* listens for connections on the named *portno* and, whenever one is made, spawns a subprocess to service the connection; the subprocess invokes the function *func*. The function *func* is called as follows:

1. Standard input, output, and error are all rerouted to the connection, so if the function reads standard input, it reads what the process at the other end has sent, and if it writes to standard output or error, it writes to the process at the other end.
2. All other file descriptors are closed.
3. There is no associated controlling terminal
4. The current working directory is "/".
5. The file creation mask **umask** is set to 0.
6. Any signal may be sent to anything desired; initially, the signal for dead child processes is set to a reaping function (which just does a *wait*(2) that returns immediately, thereby ensuring that the dead process is removed from the process table), and the hangup and signals for stopping the process from the keyboard and on input or output are ignored.

When *func* returns, the process spawned to run it exits.

RETURN VALUE

This function does not return.

If the fork to spawn the subprocess that services the connection fails, the connection is closed but no error message is given.

NAME

ne_buildserver – make the internet address of a service at a host

SYNOPSIS

```
#include net.h

struct sockaddr_in *ne_buildserver(host, service, protocol, portno)
char *host;
char *service;
char *protocol;
int portno;
```

DESCRIPTION

The function *ne_buildserver* returns a pointer to the internet address composed of the host *host* and port number *portno* offering the service *service* using the protocol *protocol*. If *portno* is present, the returned address uses that port number and ignores the *service* and *protocol* arguments.

If *host* is NULL, the local host is used. If *protocol* is NULL , the address returned will provide the requested *service*; if the *service* has only one supporting protocol (like SMTP), this will work; if there is more than one such protocol, the protocol being used will be undefined.

RETURN VALUE

On success, a pointer to the requisite internet address is returned. On failure, NULL is returned and *ne_errno* and *ne_call* are set appropriately.

WARNINGS

The return value points to a static area which is overwritten at each call.

SEE ALSO

netperror(3)

NAME

ne_ghost – make the internet address of a service at a host

SYNOPSIS

#include net.h

```
struct hostent *ne_ghost(host, address)
char *host;
struct sockaddr_in *address;
```

DESCRIPTION

The function *ne_ghost* returns a pointer to information in the host table or directory about the host named *host* or with internet address *address*.

If both a host name and an internet address are given, the data pointed to on return is associated with the named host; the internet address will be ignored unless there is no information associated with the named host. If neither a host name nor an internet address is given (that is, both arguments are NULL) the data pointed to on return is associated with the local host.

RETURN VALUE

On success, a pointer to the requisite host table entry is returned. On failure, NULL is returned and *ne_errno* and *ne_call* are set appropriately.

WARNINGS

The return value points to a static area which is overwritten at each call.

SEE ALSO

netperror(3)

NAME

netacp — accept a remote connection

SYNOPSIS

#include net.h

int netacp(fd)
int fd;

DESCRIPTION

The function *netacp* takes a socket file descriptor obtained from *netserv*(3) and blocks, waiting for a connection. It returns when a client has connected to it.

VARIABLES

Several library variables may be used to configure the system. The acceptance of a connection is made using the function pointed to by *ne_accept* (default *accept*(2)). You can change this, but unless you know exactly what you are doing it is strongly discouraged.

RETURN VALUE

On success, the file descriptor of the connection is returned. On failure, −1 is returned and *ne_errno* and *ne_call* are set appropriately.

SEE ALSO

netperror(3)

NAME

netclose – close a remote connection

SYNOPSIS

#include net.h

int netclose(fd)
int fd;

DESCRIPTION

The function *netclose* takes a file descriptor obtained from *netserv*(3), *netacp*(3), or *netconn*(3) and closes it.

RETURN VALUE

On success, 0 is returned. On failure, −1 is returned and *ne_errno* and *ne_call* are set appropriately.

SEE ALSO

netperror(3)

NAME

netconn – make a remote connection

SYNOPSIS

```
#include net.h

int netconn(service, host, protocol, portno)
char *service;
char *host;
char *protocol;
int portno;
```

DESCRIPTION

The function *netconn* establishes a connection to host *host* requesting the service *service* using the protocol *protocol*. If *portno* is present, it connects to that port number and ignores the *service* and *protocol* arguments.

If *host* is **NULL**, the local host is used. If *portno* is present, it connects to that port number and ignores the *service* and *protocol* arguments. If *protocol* is **NULL** , a connection to the named *host* will be made and the desired *service* requested. If the *service* has only one supporting protocol (like SMTP), this will work; if there is more than one such protocol, the protocol being used will be undefined.

VARIABLES

Several library variables may be used to configure the system. The connection is made using the function pointed to by *ne_connect* (default *connect*(2)); the socket is created in the domain *nso_domain* (default **AF_INET**, the Internet domain); is of the type defined by *nso_type* (default **SOCK_STREAM**, the stream socket type); and is created with the underlying protocol *nso_proto* (default 0, the default Internet domain protocols). The connection by default is set to be reused, and not to linger, this is done at the level *nso_level* (default **SOL_SOCKET**, the socket level). You can change these, but unless you know exactly what you are doing it is strongly discouraged.

RETURN VALUE

On success, the file descriptor of the connection is returned. On failure, −1 is returned and *ne_errno* and *ne_call* are set appropriately.

SEE ALSO

netperror(3)

NAME
netperror – print a network library error message

SYNOPSIS
```
#include net.h

void netperror(s)
char *s;

int ne_call;
int ne_errno;
```

DESCRIPTION
The function *netperror* takes a string *s*, prepends a message describing the last error in the network library to occur and the routine which caused it, and prints the concatenation.

The routine which caused the error is stored in *ne_call*; possible values are:

N_SOCKET	error in *socket*(2)
N_SSR1	error in *setsockopt*(2), setting reuse
N_SSL0	error in *setsockopt*(2), disabling lingering
N_CONNECT	error in *connect*(2)
N_BIND	error in *bind*(2)
N_LISTEN	error in *listen*(2)
N_ACCEPT	error in *accept*(2)
N_READ	error in *read*(2)
N_WRITE	error in *write*(2)
N_GSBN	error in *getservbyname*(3)
N_GHBNA	error in *gethostbyname*(3), *gethostbyaddr*(3)
N_CLOSE	error in *close*(2)
N_THNAME	error in *gethostid*(3) and *gethostbyname*(3)
N_GPN	error in *getpeername*(3)
N_MALLOC	error in *malloc*(3)

The number in *ne_errno* is the error number. In all cases except where the call code is N_GSBN and N_GHBNA, the number in *ne_errno* is the same as the system error number *errno*; if the call code is N_GHBNA, the value in *ne_errno* is that of *h_errno* (see *gethostbyname*(3)), and if the call code is N_GSBN, the value in *ne_errno* is one of:

N_NOSERV	no such service listed
N_NOSP	no such service/protocol pair listed

VARIABLES
All printing is done by calling the function pointed to by *ne_print* (the default is to print to the standard error).

RETURN VALUE
None.

SEE ALSO
intro(2), perror(3)

NAME

netread – read from a remote connection

SYNOPSIS

#include net.h

```
int netread(fd, buf, nchars, bufsiz)
int fd;
char buf[];
int nchars;
int bufsiz;
```

DESCRIPTION

The function *netread* reads up to *nchars* characters from the file descriptor *fd* obtained from *netacp*(3) or *netconn*(3), and stores them in the buffer *buf*. If necessary, multiple invocations of the system call *read*(2) will be made unless *nchars* is -1, in which case up to *bufsiz* characters will be read in one call to *read*(2).

RETURN VALUE

If anything is read, the number of characters read will be returned. If an EOF is encountered before anything is read, *netread* returns 0. If an error is encountered before anything is read, *netread* returns -1. If an error occurs at any time, *ne_errno* and *ne_call* are set appropriately. It is recommended you set them both to 0 before this call, and check them afterwards, since if the error occurs after at least 1 character has been read, the return value will be non-negative but *ne_errno* and *ne_call* will be set appropriately.

SEE ALSO

netperror(3)

NAME

netserv – set up a socket to receive connections

SYNOPSIS

```
#include net.h

int netserv(portno)
int portno;
```

DESCRIPTION

The function *netserv* sets up an address so that the calling process can accept connections at the port number *portno*.

VARIABLES

Several library variables may be used to configure the system. The socket is created in the domain *nso_domain* (default **AF_INET**, the Internet domain); is of the type defined by *nso_type* (default **SOCK_STREAM**, the stream socket type); and is created with the underlying protocol *nso_proto* (default 0, the default Internet domain protocols). The connection by default is set to be reused, and not to linger, this is done at the level *nso_level* (default **SOL_SOCKET**, the socket level). The maximum length of processes waiting to be *netacp*'ed is *nli_quelen* (default 1). You can change these, but unless you know exactly what you are doing it is **strongly** discouraged.

RETURN VALUE

On success, the file descriptor of the socket is returned. On failure, −1 is returned and *ne_errno* and *ne_call* are set appropriately.

SEE ALSO

netperror(3)

NAME

netwrite – write to a remote connection

SYNOPSIS

```
#include net.h

int netwrite(fd, buf, nchars)
int fd;
char buf[];
int nchars;
```

DESCRIPTION

The function *netwrite* writes up to *nchars* characters to the file descriptor *fd* obtained from *netacp*(3) or *netconn*(3), obtaining them from the buffer *buf*.

RETURN VALUE

On success, the number of bytes successfully written is returned. On failure, -1 is returned, and *ne_errno* and *ne_call* are set appropriately.

SEE ALSO

netperror(3)

NAME

offhostname – return official host name of a host

SYNOPSIS

#include net.h

char *offhostname(host)
char *host;

DESCRIPTION

The function *offhostname* returns the official host name of the argument *host*. Here, *host* must be a name and not Internet numbers.

RETURN VALUE

If the host name is not found in the database, NULL is returned and *ne_call* and *ne_errno* are set appropriately.

WARNING

The return value is contained in a static buffer which is overwritten by each call.

SEE ALSO

netperror(3)

NAME

seal, unseal – digitally sign, and optionally encrypt, messages

SYNOPSIS

#include seal.h

```
char seal(locproc, lochost, remproc, remhost, buf,
char *locproc, *lochost;
char *remproc, *remhost;
char buf[];
int bufsz;
rover_to msg;
unsigned int *flag;
```

```
char unseal(locproc, lochost, remproc, remhost, buf,
char *locproc, *lochost;
char *remproc, *remhost;
char buf[];
int *bufsz;
rover_to msg;
unsigned int *flag;
```

DESCRIPTION

The function *seal*() takes the message contained in *buf* and of length *bufsz* (maximum **DATASZ**), and writes a specially formatted message into *msg* containing the originating process (or user) *locproc*, the originating host *lochost*, the destination process (or user) *remproc*, and the destination host *remhost*. This packet is cryptographically signed using (for *seal* and *unseal*) the key associated with *lochost*@*locproc* or (for *rseal* and *runseal*) the key associated with *remhost*@*remproc*.

For *seal* and *rseal*, users may request two special options by setting the bits in *flag* appropriately:

F_NONE clear all bits
F_ENCRYPTencrypt the message
F_PERMESSAGEmake a new connection for each packet

These are to be or'ed together. The last flag is useful in conjunction with the *rover*(1) digital signature scheme; normally, that system keeps the first connection open. The flag instructs *rover* to drop the connection after authenticating each packet.

The password is obtained by checking for a series of files, and if none are present, prompting at the controlling terminal. Let *$HOME* be the user's home directory, *proc* be argument 0 of the process (that is, the basename of the program executed), and "<SP>" the space character (octal 040, hex 0x20). Then the files

$HOME/..word.*proc.host*<SP>
$HOME/..word.*proc*<SP>
$HOME/..word<SP>

are checked for, in that order. If any exists, is owned by the real UID of the process, and is readable by the owner only, its contents are used as the password. If none of those files meet the criteria, a prompt for the password is sent to the controlling terminal.

RETURN VALUE

All routines return a nonzero code indicating the result of the sealing or unsealing. To understand these, one must realize that the paradigm is that the local process will seal the message using its key and send it to *rover*, which will then unseal the message and reseal it using the destination's key, and return the newly-sealed message to the originator. The originator then forwards the message to the destination. Hence, the result is returned as the logical or of the following:

F_PERMESSAGEconnections on a per-message basis

F_ENCRYPTencrypt the message
F_EOF unexpected end-of-file encountered
E_NOORIGorigin password unavailable
E_NODESTdestination password unavailable
E_BADINTintegrity check failed; corrupted message
E_STALEmessage older than **ROVER_INTERVAL**
E_GARBLEDmessage is garbled
E_DBADINTintegrity check failed; corrupted message
E_DSTALEmessage older than **ROVER_INTERVAL**
E_DGARBLEDmessage is garbled

E_BADINT , **E_STALE** , and **E_GARBLED** refer to the message as unsealed by *rover*; **E_DBADINT** , **E_DSTALE** , and **E_DGARBLED** refer to the message as unsealed by the destination. Note that error flags may be placed within the message itself, but all such flags are included in the digital signature.

BUGS

The use of the password files is **strongly** discouraged, but for processes without a controlling terminal and no cryptographic box, you're stuck.

NAME

dbm – rover database format

SYNOPSIS

dbmfunc.o

DESCRIPTION

The command *dbm*(1) builds and manages a database of cryptographic information for the digital signature scheme *rover*. The format of each entry in the database is:

```
char  inuse;
char  proc[66];
char  host[66];
char  key[1025]
int   keylen;
```

The first field indicates whether the item is active or has been deleted. The second indicates the name of the process (user), the third, the host on which the process (user) executes, the fourth, the cryptographic key; and the fifth, the number of bytes in the key.

SEE ALSO

dbm(1), rover(1)