The Fast Encryption Package

Matt Bishop

August, 1988

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Memorandum 88.3

NASA Cooperative Agreement Number NCC 2-398

RIACS

Research Institute for Advanced Computer Science

strongly recommended you use the library with the password checker; the gain in speed is considerable.

Each part of the package can be configured and installed separately, or all can be configured and installed at once. We shall discuss configuration and installation of the entire package; in the process how to configure each part will be detailed. In any case, it is wisest to configure everything so that the parts can all be installed with minimum effort later, if need be.

2. Configuring and Compiling the Package

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

1. Determine whether your system is closer to System V or Berkeley UNIX and copy the appropriate file (**Make.bsd4** or **Make.sysv**) to **Makefile**.
2. Edit **lib/include/mach.h** to add the characteristics of your machine if it does not already contain them. The next section describes how to change these compile time macros.
3. Edit **passwd/sys.h** to set up the characteristics of your system for the password changing program if it does not already contain them. Section 4 describes how to set these compile-time macros.
4. Edit **Makefile**. The parameters which may have to be reset are described in section 5.
5. Switch to the superuser and type

   make install

   to install the relevant libraries, programs, and manual pages.
6. If the password checker has been installed, configure its data files. Section 6 describes how to do this.
7. If the password checker has been installed, set it up to run every so often. Section 7 describes one way to have it do so.
8. If the password changer has been installed, set up its configuration file. Section 8 describes how to do this.
9. Now relax; you deserve one calm, placid night before you get the report of users with known passwords, or before users assault you because they can no longer use their login name as a password!

3. Include File

There are six parameters that control how the libraries are compiled. These are described below. All changes should be made in **lib/include/mach.h**. The changes depend on the hardware on which the programs are to run, so the compilation sequence requires this dependence to be reflected in the file. The last subsection of this section describes an aid to setting these.

First, start up an editing session on **include/mach.h**. Set up a conditional compilation sequence for your computer; that is, put a

```
#ifdef yourcomputer
#endif
```

where *yourcomputer* is the constant corresponding to your computer. This should *not* be "unix" but rather something that uniquely specifies the system on which the code will be running. For example, on VAX machines, an appropriate constant would be "vax". If necessary, use the host name and add that to the C compiler options in the top-level Makefile. Some systems have already been configured; they are shown in the following table.

Preconfigured Systems	
constant	*system*
CRAY2	CRI Cray 2 running UNICOS
convex	Convex C-1 running Convex UNIX
m68k	Apple Macintosh II running AOS/UNIX
m68000	SGI IRIS 2500T/3030 running GL2-W3.6
sequent	Sequent Balance 21000 running Dynix 2.1
sun	Sun series running 4.2 BSD UNIX Release 3.3
uts	Amdahl 5880 running UTS
vax	VAX 11 series running 4.3 BSD

All of the following constants should be in "#define" lines after the "#ifdef" line but before the "#endif" line, since they vary from computer to computer.

3.1. AUTOINC

Define this constant if your machine architecture supports autoincrement addressing mode, and your C compiler will use it to increment pointers. This forces the use of autoincrement mode to step through the bit permutation arrays. Do not set this if no autoincrement addressing mode exists because this will generate a move and an add, as opposed to a move.

3.2. BITSPERWORD

Set this to the number of bits per word in your computer. It should be the number of bits allocated to the type **WORD** described below.

3.3. FIELDS

Define this constant if using the field syntax of C produces faster code than the shift-and-mask alternative. This (usually) causes the compiler to produce code that uses bitfield extraction instructions.

3.4. FORMAT

Define this constant to be a format suitable for handing to *printf*(3) to print a constant as an array element with the declaration **WORD**. (You must include the surrounding double quotation marks.) For example, if a word is declared as "unsigned int long", this would be defined as "'"\t0x%x,\n"'". Choose something your compiler finds appropriate.

3.5. NETORDER

This should be defined if the byte ordering is the same as network byte ordering. If you have defined **FIELDS**, you probably do not have to define this one, because the code will probably use bitfield extraction instructions which handle byte ordering properly, rather than equivalent shift-and-mask instructions, to access parts of words; But as this is not guaranteed, you would be wise to do so. It is also a safety measure just in case you decide to turn off **FIELDS** later.

3.6. STRUCTFROMTOP

This should be defined if **FIELDS** is defined and the computer allocates structure elements from the top to the bottom. As with **NETORDER**, if **FIELDS** is not defined this constant probably need not be defined since structure accesses should not be used to access parts of a word (and hence the order of the fields in the structure does not matter); but you should define this constant anyway if appropriate.

3.7. WORD

This should be set to the C type that allocates one machine word of storage. On most machines this type will be an *unsigned long int*; however on some machines it may be a non-portable declaration (for example, on the Convex C-1, to use 64 bit words, **WORD** must be defined as *unsigned long long int*).

3.8. Examples

The following is a sample definition for a VAX running UNIX

```
#ifdef vax
#       define AUTOINC                 /* use autoincrement address mode */
#       define BITSPERWORD 32          /* 32 bits per word */
#       define FIELDS                  /* quicker to access bit fields */
#       define FORMAT "\t0x%lxL,\n"    /* write a WORD */
#       define WORD unsigned long      /* 32 bits */
#endif
```

The VAX has an autoincrement mode and its bitfield access instructions are quicker than shifting and masking bits. The type "unsigned long" allocates one word of storage, which is 32 bits long, and the contents of that word can be printed in hexadecimal using the format string '"\t0x%lxL,\n"'. Note the "\t" and the trailing '\n'; these make the files containing the array definitions easy to read. The "," is necessary because the number is one in a sequence of numbers used to initialize an array, and the "L" indicates the initializer is of type long. Notice that the VAX orders its bytes from right to left, so **NETORDER** is not defined, and the 4.3 BSD UNIX C compiler arranges structure fields in reverse order, so **STRUCTFROMTOP** is not defined.

The Convex C-1 does things differently. It has an autoincrement mode, orders bytes from left to right, and allocates structure fields in the same way as the VAX; hence **AUTOINC** and **NETORDER** are defined but **STRUCTFROMTOP** is not. It also has a piplining mode in which two 32 bit words may be combined to form a 64 bit word; but in this case, the bitfield extraction instructions cannot be used because they would cross the 32 bit word boundary. So, **FIELDS** is not defined, and **BITSPERWORD** is set to 64. These 64 bit words must be declared as "unsigned long long int", are printed using the special notation "%llx", and constants of that type must have "LL" on the end. This

leads to the following declaration block.

```
#ifdef convex
#       define  AUTOINC                         /* autoincrement mode */
#       define  BITSPERWORD 64                  /* 64 bits per word */
#       define  FORMAT "\t0x%llxLL,\n"          /* write a WORD */
#       define  NETORDER                        /* bytes are in network order */
#       define  WORD unsigned long long int     /* 64 bits */
#endif
```

3.9. Help and Summary

As an aid to determining the best values for these constants, there is a program that will make various tests and suggest possible values. Set the makefile variable ${*SYSTYPE*} as described in the next section, and then type

make -s defines

The output will look like

```
        /* these assume a word is declared as unsigned long int */
        /* if not, any or all of the following may be bogus */
#define WORD unsigned long int/* NOT EVEN A GUESS */
#define FORMAT "\t0x%lxL,\n"            /* NOT EVEN A GUESS */
#define BITSPERWORD         32      /* 32 bits/unsigned long int */
        /* the next two should be correct for this machine ... */
#undef  STRUCTFROMTOP       /* this should be correct */
#undef  NETORDER            /* bytes not in network order */
        /* checking FIELDS ... this will take about two minutes */
#define FIELDS                      /* this should be right */
        /* checking AUTOINC ... this will take about two minutes */
#undef  AUTOINC             /* WARNING -- too close to be sure */
```

This suggests some possible values for the constants.

The first three macro definitions assume that the type of storage that allocates 1 word is ''unsigned long int''. If this is false, all three definitions are probably wrong; otherwise, they will be correct. This program was run on a Sequent and on that machine, they are correct. (But on the Convex using 64 bits per word, this program would give the same results as on the Sequent because on the Convex, the proper type is ''unsigned long long int''.) The next two definitions are almost always right; here, they indicate that neither **STRUCTFROMTOP** nor **NETORDER** should be set. The last two take about two minutes each to check, and involve timing some loops. They should be used as guidelines, because they are very sensitive to system load; for example, for the Sequent, the settings shown happen to be best, but when the timings were taken for **AUTOINC** the results were too close to be used without question. (This is when going to the architecture manual and checking the code the compiler produces for the statement

x = *p++;

is a good idea.) Note that the program warns you of this in the comment field.

The following table summarizes the constants that can be set and gives sample values.

machine/mach.h file constants		
field name	*sample value*	*what*
AUTOINC	1	autoincrement addressing mode
BITSPERWORD	32	bits per *WORD*
FIELDS	1	use fields
FORMAT	"\t0x%lxL,\n"	print word-length constants
NETORDER	1	bytes in network order
STRUCTFROMTOP	1	structs from top to bottom
WORD	unsigned long	stores (machine) word

4. Password Changing Macros

These changes are to be made in **passwd/sys.h**. These are relevant only to the password changer, so they need not be made unless that program is to be installed. Each system has its own methods of handling the password files; some allow the user to set the GECOS field, some do password aging, and so forth. This section describes each relevant constant and how it might be set.

4.1. CHFN

If your machine uses the *passwd*(1) command to allow users to change their GECOS information, define this constant. One way to determine if this should be set is to look for a command called *chfn*(1); if that command exists and is a link (or alias) to *passwd*(1), it should be set. The GECOS information is in the fifth field of the password file, and usually includes the user's name, office, and telephone extension.

4.2. GETUSERSHELL

If your machine uses the *passwd*(1) command to allow users to change their login shell, define this macro to call the function returning valid user shells. The assumed calling sequence is that no arguments are passed, and each call returns either a string with the full path name of a valid user shell, or NULL meaning the end of the list. One way to determine if this should be set is to look for a command called *chsh*(1); if that command exists and is a link (or alias) to *passwd*(1), it should be set. The name of the login shell is in the seventh field of the password file, and usually must be chosen from a list of shells in a system file.

4.3. AGE_FIELD

Set this if the password structure in the manual has the field "pw_age" defined. This field is appended to the password field (field 2) and separated from the encrypted field by a comma. To decide if this should be set, either look at *getpwnam*(3) in the manual for that field, or check the password file for lines with a second field consisting of 13 characters followed by a comma and two more characters; if such a line is present, define this.

4.4. FGETPWENT, SETPWFILE

These functions enable password files other than the default to be accessed, and these need not be defined if only the default password file will be used. ${FGETPWFILE} is a macro that assumes its definition is a function called with a pointer to the password file; on each call, it returns the *passwd*(5) structure of the next user in the file, or NULL if the end of the file has been reached. ${SETPWFILE} is a macro that assumes its definition is a function called with the name of the password file; it returns nothing, but once called, *getpwnam*(3) and *getpwuid*(3) use that file rather than the default to obtain data associated with users. If both are defined, ${FGETPWENT} is used. If neither is defined, users can only change information in the default password file.

4.5. SYSLOG

If the library logging system *syslog*(3) is available, define this; if those functions are available but require some library other than the default C library, set this macro to the name of that library; if these functions are not available, don't define it. These are useful for logging things.

4.6. DBMLIB

Some systems such as 4.3 BSD maintain a hashed version of the password file to speed lookups. These versions are usually maintained using the *dbm*(3) or *ndbm*(3) functions (the two are the same so far as this use is concerned.) Define this macro if the password file is also stored using *dbm*(5) format; if that format is used and this macro is undefined, or if that format is not used and this macro is defined, the program will fail miserably. Set the value to the name of the library containing these functions; if that is the standard C library, you need only define the macro without setting it to anything. To determine if you must set this, look for two files in the same directory as the password file and having the extensions ".dir" and ".pag". If they exist, you have to set it.

4.7. UID_TYPE

This is the type returned by the library function *getuid*(2). It may be set to *int* on all machines this program has been ported to (so far), but it should be set properly to keep *lint* happy. Look at the type of *getuid*(2) in the manual to find the proper setting.

4.8. RENAME

If your system has a routine that renames a file atomically, define this to be that routine; the first argument is the old name and the second the new name. If this is not defined, a sequence of *unlink*(2) and *link*(2) system calls will be used to acheive the same effect. The problem with this is that should the program be interrupted (by a system crash, for instance) during this sequence of system calls, the password file may not exist under the correct name. If this happens, the system must be booted in single-user mode and the temporary file renamed manually before switching to multi-user mode.

4.9. OPENLOG

Define this only if ${*SYSLOG*} is defined. The macro has three arguments; its definition has either two or three arguments, depending on what your system has (see *openlog* (3) for the right number.) If the right number is two, simply omit the last argument in the macro definition.

4.10. GETDOMAIN

This is defined to be a function of two arguments. The first argument is a character array where the domain is to be defined; the second, the number of character spaces in the first argument. It returns the domain name of the host. If no such function is available, do not define it; in this case the host name will be obtained using ${*GETHOST*} or ${*HOSTNAME*} and everything after the first period "." will be used as the domain name.

4.11. GETHOST

This is defined to be a function of two arguments. The first argument is a character array where the domain is to be defined; the second, the number of character spaces in the first argument. It returns the fully qualified (domained) name of the host. If no such function is available, do not define it.

4.12. HOSTNAME

If ${*GETHOST*} is undefined, define this to be the fully qualified (domained) name of the host. For example, the host **prandtl** would have this set to **prandtl.nas.nasa.gov**. This should not be set if ${*GETHOST*} is defined.

4.13. ALLOWCORE

If set, *passwd* will generate a core dump should an appropriate signal be caught. (See *signal* (2) or *sigvec* (2) for a list of these signals.) **WARNING:** *This option should never be set for production; it is used only to debug the program, and creates a potential security hole in your system.*

4.14. ROOTID

If set, *passwd* will assume the numeric value of this macro is the numeric UID of the superuser. It is used to allow a systems programmer debugging the program to change someone else's password. **WARNING:** *This option should never be set for production; it is used only to debug the program, and creates a potential security hole in your system.*

4.15. Example

Here are sample settings for a vanilla BSD 4.2 system:

```
#ifdef  BSD4_2
#       define  CHFN
#       define  GETUSERSHELL()          getusershell()
#       define  RENAME(old,new)         rename(old,new)
#       define  OPENLOG(a,b,c)          openlog(a,b)
#       define  SYSLOG
#       define  UID_TYPE                int
#endif
```

Here, both *chfn* (1) and *chsh* (1) are to be defined, the library logging system *syslog* (3) is defined in the standard C library and *openlog* (3) has two, not three, arguments; there is an atomic *rename* (2) function, and *getuid* (2) returns an int.

For the UTS operating system running on **prandtl**, an appropriate definition scheme would be:

```
#ifdef  SYSV
#       define  AGE_FIELD
#       define  FGETPWENT(x)            fgetpwent(x)
#       define  HOSTNAME                prandtl.nas.nasa.gov
#       define  UID_TYPE                unsigned  short
#endif
```

Prandtl is very close to a System V system, and the definitions above reflect this. Note that the host name is compiled in, and the domain name will be derived from it (it will be **nas.nasa.gov**.)

The following table summarizes the constants that can be set and gives sample values.

passwd/sys.h file constants

field name	*sample value*	*what*
AGE_FIELD	1	password aging implemented
ALLOWCORE	1	allow core dumps (**DANGER**)
CHFN	1	enable *chfn*(1) function
DBMLIB	-ldbm	*dbm*(3) library
FGETPWENT	fgetpwent	entry from alternate password file
GETDOMAIN	getdomainname	get domain name
GETHOST	gethostname	get host name
GETUSERSHELL	getusershell	list legal login shells
HOSTNAME	hydra.riacs.edu	host name
OPENLOG	openlog	begin *syslog*(3)ging
RENAME	rename	atomic file rename function
ROOTID	0	uid of superuser
SETPWFILE	setpwfile	use alternate password file
SYSLOG	syslog.o	*syslog*(3) functions
UID_TYPE	int	type of *getuid*(2)

5. Makefile

These changes are to be made to the Makefile in the top-level directory. Make the ones to **lib/include/mach.h** *first* since an aid to setting some of these requires that the libraries compile. (See the last subsection of this section for details.)

5.1. ADMIN

Set this to the list of users who are to receive copies of the output of the password checking system when no-one specific is named. This should be set to a specific person or mailing alias, not to something generic like *root*, because the output lists users and passwords when they are found.

5.2. BINDIR

This is the name of the directory in which executable programs and shell scripts are to be placed.

5.3. CHKFILE

By default, the password tester checkpoints itself using a checkpoint file in the ${*MISCDIR*} directory with a base name of "*mmddyy*"; where *mm* is the month (01 – 12), *dd* is the day of the month (01 – 31), and *yy* is the last two digits of the year. Set ${*CHKFILE*} to what you want the base name to be if you want something else. NOte that the password checking mechanism creates two files with this base name; one has the extension ".pwd" and the other the extension ".dic".

5.4. CHKTIME

The password tester is instructed to checkpoint itself every so often; this preserves work if the system should crash. By default the password checking system checkpoints itself every 600 seconds; to change this, reset ${*CHKTIME*} to the number of seconds between checkpoints. Note that the minimum value of ${*CHKTIME*} is 300.

5.5. COPTS

This is a list of options passed to the compiler when the programs are generated. It is *not* passed to the compiler when the library is generated. Currently, there are only two flags of interest, and these apply only to the password checker: "–DUSE_FULL_CRYPT", which forces the interface to the standard *crypt*(3) function to be used (which is a bit slower than the more compact interface, but allows the program to be run using the system's version of *crypt*), and "–DCANT_ASSIGN_ALL_FIELDS", which means that the compiler does not allow structure assignment, but instead each field must be copied separately.

5.6. CRY, DES, SHC

These are the suffixes dictating which versions of the libraries are to be used. There are three different main routines making up the library: the *DES* subroutines, the *crypt* (CRY) subroutines, and the *short crypt* (SHC) subroutines. Each one may be one of four different configurations; each of the four configurations is identified by two parameters. The first parameter is the number of bits handed to the S boxes, and the second is the

number of permutations used to generate the key schedule (if you don't understand these terms, don't worry.) Each of these parameters has two possible settings: the number of bits handed to the S boxes will be either 6 or 12, and the number of permutations used to generate the key schedule will be either 1 or 3. Thus, the four configurations of each library are identified by the names *061*, *063*, *121*, and *123*. Setting **DES** to any of these four values causes the appropriate configurations of the DES subroutines to be loaded into the library; similarly, **CRY** indicates the configurations of the *crypt* subroutines to use, and **SHC** the configurations of the *short crypt* subroutines to use.

For guidance on setting these, there is a program that will run a series of tests and suggest values. See the last subsection of this section.

5.7. DESLIB

This is the full path name of the fast encryption library. The library's basename must be **libdes.a** and the directory should be the same as ${*LIBDIR*} unless the library was compiled and installed separately.

5.8. DESLINT

This is the *lint*(1) file generated from the fast encryption library. Currently it should be left blank; someday, such a library will be supplied and might actually persuade *lint* to shut up.

5.9. DICTDIR

This is the directory containing word lists used as guesses by the password checker.

5.10. INCDIR

This is the name of the directory that the include file **des.h** is to be put in.

5.11. LIBDIR

This is the name of the directory that the library is to be put in. It will be installed as **libdes.a** in this directory.

5.12. LOCALLIB

Many of the executables in this system use the argument parsing function *getopt*(3), which may not be in the standard library. If not, set this to the name of the file or library containing the object version of this routine.

5.13. MAILER

This is the program that is used to mail the results of the password testing to the appropriate people. It is invoked by giving the list of addresses, separated by blanks, as command-line arguments, and the letter as standard input. If possible, a "Subject: " field may be specified using an "–s" option; see **SUBJECT**, below. procedure.

5.14. MISCDIR

This directory contains several files used by the password checker to determine where to get current password files and word lists as well as letters to send to users whose accounts have no password or whose passwords have been guessed.

5.15. PWDDIR

This is the directory in which the password files of the hosts being checked are kept. The full password files as well as incremental changes to them are kept here.

5.16. PWEXEC

This is the name under which the password changing program is to be installed.

5.17. PWFILE

This is the name of the file in which users' encrypted passwords are stored. ${*PWEXEC*} will change the passwords (and possibly the shell and GECOS fields) in it.

5.18. PWLOG

This is the name of the log file for the password changer. See section 7 for details.

5.19. PWTEST

This names the file describing allowable user passwords. It is described in more detail in section 7.

5.20. PWTYPE

This is the parameter describing the type of the system so far as the password changing program should know. It usually must be more specific than the generic ${*SYSTYPE*} variable; for example, valid values are BSD4_2, BSD4_3, SUN, and SYSV. If none of these are *identical* to your system, look in section 3 to set up your own type, and set this to the constant that will include your definitions.

5.21. ROFF

This is the version of *troff*(1) that your site uses. Manual pages (as well as this guide) are printed with it.

5.22. ROOTDIR

This is the name of the directory that the password checking system and its associated programs are to be placed in. It will be the root of a tree of directories used by that checker. is to be installed in.

5.23. SUBJECT

The legal settings of this field are "yes" and "no". If the mailer allows specification of a "Subject: " header field by giving a command-line option of the form "–s "*subject*"", set this to "yes". Then the first and second command-line arguments will be set to "–s "*subject*"", and the addresses will be the third (and successive)

arguments.

5.24. SYSTYPE

Set this to "BSD4" if your machine uses the 4.2 or 4.3 Berkeley timing mechanisms; set it to "SYSV" id your machine uses the System V timing mechanisms. This constant is used to compile various programs that time the routines and help you choose the fastest for your system.

5.25. TBL

This is the version of *tbl*(1) that your site uses. It is used to print this guide.

5.26. TMPDIR

The password checker generates many temporary files; this is the directory in which those files are placed.

5.27. VERSION

This describes the version of the library to make. Each version is in a subdirectory of the directory **lib** named **des***computer* (for example, the VAX version is **lib/desvax.**) If there is no version corresponding to your computer, use one of the generic versions. Generic versions are located in the directorues named **des***nn*, where *nn* is the maximum number of bits per word. Currently only 32 and 64 bit words are supported (in subdirectories **des32** and **des64**, respectively.) To determine which to use, use the following rule of thumb: if your machine's word size is 48 bits or more, it is safe to use the 64 bit package; if your machine's word size is 24 bits or more, it is safe to use the 32 bit package; if your machine's word size is under 24 bits, this package won't work. If your machine has a wordsize of 96 bits, the package could be optimized even more; if you are willing to allow the author to use your machine to generate such a package, please let him know!

5.28. Help and Summary

One important question is how to choose the best values for the makefile variables ${*DES*}, ${*CRY*}, and ${*SHC*}. There is a program, **lib/testing/whichone**, which will generate recommended values. To execute it, set the makefile variables **SYSTYPE** and **VERSION** correctly, and type

```
make -s recommend
```

(omit the "–s" if you want to see the commands as they are executed.) After quite some time, this will generate output of the form

```
DES=121
CRY=123
SHC=123
```

These are the recommended configurations for your system.

As with the recommendations for **AUTOINC** and **FIELDS**, these values are subject to the vagraties of system loads. The program decides which configurations to recommend by compiling all four configurations of each routine, and timing 10 runs each for 10 (virtual) seconds; it computes an average time per run from this data, and from these

averages suggests an appropriate setting for each routine. The timings are a pretty reliable indicator of what is best, but it is recommended you run this during off hours or when the load is relatively constant.

The following table summarizes the makefile variables that can be set.

Makefile file constants		
field name	*sample value*	*what*
ADMIN	pwlist	who gets output
BINDIR	${*ROOTDIR*}/=bin	where to put executables
CHKFILE	*none*	checkpoint file's base name
CHKTIME	600	checkpoint interval (seconds)
COPTS	*none*	compile options
CRY	123	config of *crypt*
DES	123	config of DES
DESLIB	${*LIBDIR*}/libdes.a	path name of library
DESLINT	*none*	*lint* library file
DICTDIR	${*ROOTDIR*}/=dict	where to word lists
INCDIR	${*ROOTDIR*}/=include	where to put include files
LIBDIR	${*ROOTDIR*}/=lib	where to put **libdes.a**
LOCALLIB	*none*	library containing *getopt*(3)
MAILER	/usr/ucb/Mail	program to mail results
PWDDIR	${*ROOTDIR*}	where to put password files
PWEXEC	/bin/passwd	password changing program
PWFILE	/etc/passwd	default password file
PWLOG	/etc/passwd.log	password log file
PWTEST	/etc/passwd.test	password restriction file
PWTYPE	BSD4_3	type of password file
ROFF	psroff	version of *troff*(1)
ROOTDIR	/usr/local/adm/pwcheck	root of password system
SHC	123	config of short *crypt*
SUBJECT	yes	does MAILER know "–s"
SYSTYPE	BSD4	use 4.2 BSD timing functions
TBL	tbl	version of *tbl*(1)
TMPDIR	${*ROOTDIR*}/=tmp	where to put temporary files
VERSION	des32	version of library

6. The Password Checker

The password checking system consists of a number of shell scripts and programs that allow a system administrator to check passwords against a dictionary. The package is installed during the procedure in section 2. This section describes how to set up and run the package.

The password checking package requires two sets of data: the password files to be checked and a set of dictionaries to check them against. A password file is obtained from the system being checked by copying **/etc/passwd** (or some other appropriate file); a dictionary is generated from a word list file as described below. In addition, assorted miscellaneous files tie these all together.

First, we will describe each of the required files; then we shall describe the command used to check passwords. An appendix summarizes the possible error messages.

6.1. Word Lists

A word list is simply a file containing a list of words, one per line. Comments may be interspersed by putting them on a line with a sharp "#" character in column 1. Words in these files are used to generate a dictionary by the shell script ${*MISCDIR*}/exec.list.

6.2. Dictionaries

The password checking system uses the words in a dictionary as possible passwords; it encrypts them and compares them to the encrypted passwords. Typically, dictionaries are derived from word lists, but users may supply their own; in this latter case, blank lines are deleted, words are truncated to 8 cheracters or less, and the words are sorted before being merged into a system dictionary. As with word lists, dictionaries have one word per line.

6.3. ${*MISCDIR*}/exec.list

A word list is turned into a dictionary by performing some transformation on the list, truncating words to 8 cheracters, sorting the result, and pruning duplicates. This file, which must be a Bourne shell *sh*(1) script, takes the word list name as its single argument, and possibly the debugging flags –x and/or –v as options. The program should produce a new word list wit one word per line. (Sorting and truncation to 8 characters is done later.) The default script will use the word list, its words spelled backwards, and change cases of characters.

Note that the script must be of the Bourne variety. If desired, this file may call another program. So, for example, to process word lists by using the C-shell script cwords, this file should contain

```
:
csh cwords $*
```

6.4. ${*MISCDIR*}/exec.pwds

A password file encodes information about users, such as account names, personal names, and so forth, that often is used as passwords. This file, which must be a Bourne shell *sh*(1) script, takes the password file name as its single argument, and possibly the debugging flags –x and/or –v as options. The program should produce a new word list wit one word per line. (Sorting and truncation to 8 characters is done later.) The default script will use the name and GECOS field of the password file to generate a list of user login names and personal names, ch it then runs through ${*MISCDIR*}/exec.list.

Note that the script must be of the Bourne variety. If desired, this file may call another program. So, for example, to process word lists by using the C-shell script cpwds, this file should contain

```
:
csh cpwds $*
```

6.5. ${*MISCDIR*}/wordlists

This file contains a list of files containing word lists; these word lists are used to generate the dictionary used to test passwords. The path names should be absolute path names, not relative ones; comments may be placed on lines with a sharp "#" in column 1 or after the path names and with a tab in front of them (no, blanks won't work.)

6.6. ${*DICTDIR*}/sysdict

This is the dictionary that is used to check passwords. It is made by processing the files of word lists in ${*MISCDIR*}/wordlists. When the password checker runs, it first checks to see if any of those files have been changed since sysdict was last modified; if so, sysdict is updated to reflect this fact. If not, sysdict is not changed. If sysdict does not exist, it is created.

Note that unless told not to, the password checker will add to sysdict a word list made up of information from the password files. This allows more complete checking to be done.

6.7. ${*MISCDIR*}/pwd.found, ${*MISCDIR*}/pwd.none, ${*MISCDIR*}/pwd.illegal

These files contain programs to mail messages that can be mailed to users if the password checker determines their password, that they have no password, or that the the password cannot be typed, respectively. A sample file looks like:

```
cat << \xxEOFxx | $MAILER %1
Hello,
  The password to your account "%1" is "%2". Considering
how easily I was able to guess it, this password does not provide
much protection against an unauthorized person using your account.
You should change it at once to something more difficult to guess,
such as your mother's maiden name mis-spelled backwards.
  Please contact the consultants if you have any questions.

                    Thank you,
                      System Daemon
xxEOFxx
```

Within these files, the string "%1" will be replaced by the user's account name in the form *login@host*, enabling the precise account to be identified; the string "%2" will be replaced by the password (if one exists) or by nothing (if not).

The contents of these files may be changed as appropriate. The shell variables SUBJECT and MAILER are defined in the environment when this is run. The contents of the file are executed as input to the Bourne shell *sh*(1) after the "%" substitutions are made.

6.8. ${*MISCDIR*}/nocheck

This file lists specific users, one per line, whose passwords are not to be tested. This turns off *all* password cracking for that user, including the check for an empty password. The account names must be in the same form as the recipient address, that is

login@host.

6.9. ${*MISCDIR*}/nonotify

This file lists specific users, one per line, who are not to be notified if their password is found or they have no password, and the option to notify users of these events was given. The account names must be in the same form as the recipient address, that is *login@host*. The purpose of this file is to allow systems to have accounts such as *guest* which have no password by design and not send warning mail to those accounts. In all cases, the system will notify the ${*ADMIN*}.

6.10. ${*MISCDIR*}/machines

This file contains a list of hosts whose password files are to be checked. Each line contains one host; if the password file is other than the usual /etc/passwd, the line has the form *hostname:password_file*. (for example, *icarus:/usr/etc/passwds*). If no colon ":" occurs in the line, the word is taken to be the name of a host unless the character "/" appears, in which case it is taken to be the path name of the password file on the local host. The path name should be an absolute path name, not a relative one; comments may be placed on lines with a sharp "#" in column 1 or after the path names and with a tab in front of them.

For example, in the following file

```
icarus
/usr/pwds
prandtl:/x/zork
```

the files /etc/passwd on **icarus**, /usr/pwds on the local host, and /x/zork on the host **prandtl** will be analyzed.

The format of the names given is restricted only by what *cp* (1) and ${*REMCOPY*} will recognize and use.

6.11. ${*PWDDIR*}/*hostname*, ${*PWDDIR*}/*hostname*.old

These files contain the password file entries for the named host, and the previous password file for the named host. Note that each password file is given the host's name; this is vital since the user notification option will not work otherwise! Also, when the password checker runs, all password files are amalgamated into one; this allows certain optimizations that otherwise could not be made.

7. Running the Password Checker

The actual checking program is **bin/testpwds**. The interface is very simple but not very user-friendly, because it assumes that it has access to dictionaries and password files (see *testpwds* (8) for details.) The script **runcheck** provides a much cleaner interface; it generates dictionaries, updates password files, and runs *testpwds* automatically.

Runcheck has several options:

-a This option lists the names of people to whom the results of the audit are to be sent. If this option is given, no mail is sent to the address named in ${*ADMIN*}. As an example, the option

–aauditor,audit@domain

sends the results of the check to **auditor** and **auditor@domain**.

–d This option uses file *filename* as a dictionary, in addition to any generated from the word lists listed in ${*MISCDIR*}/**wordlists**.

–f Normally, if no passwords have changed since the last time the password file was updated, **runcheck** reports this and quits. This option forces **runcheck** to carry out the checking anyway.

–i Normally, **runcheck** does not report users with illegal passwords. This option instructs it to do so, and if the **–u** option is given, the appropriate users are also notified.

–n Normally, **runcheck** updates password files only when something has changed. This option prevents the updating and indeed even the checking for changed entries. It directs **runcheck** to use the password files already in **pwd**.

–p Normally, **runcheck** adds information gleaned from the password files to ${*DICTDIR*}/**sysdict** before using it; this option prevents that. If some word list has changed, though, ${*DICTDIR*}/**sysdict** will be updated to reflect that.

–r This option causes **runcheck** to restart the checking of the last run. The run's checkpoint files are presumed to have the basename *name*, which should be a full path name. (Normally, checkpoints go into the directory ${*MISCDIR*} and are named as indicated in section 4.3.) Any other host names will be ignored; it is best to give this option and no others.

–u With this option, two types of letters are sent to those users who are not in ${*MISCDIR*}/**nocheck** and whose password **runcheck** has found or whose password **runcheck** has determined is empty. If the password is known, the file ${*MISCDIR*}/**pwd.found** is run through a program that substitutes the mail address for "%1" and the password for "%2", and then executes the contents of the file as a Bourne shell command (which should result in a message being sent to the user; see the previous section); if there is no password, the file ${*MISCDIR*}/**pwd.none** is run through a program that substitutes the mail address for "%1" and then executes the contents of the file as a Bourne shell command (which should result in a message being sent to the user; see the previous section).

A typical command to check passwords would be:

runcheck –u –amab@riacs.edu

This notifies users who have guessable passwords and sends a complete list of accounts and passwords (as well as accounts with no passwords) to *mab@riacs.edu*.

runcheck –r/usr/local/crypt/misc/070476

This restarts an interrupted run; the last checkpoint of the run was made on July 4, 1776 (yes, computer crackers *have* been around a long time.)

It is recommended you run a complete check when you install the system, then once a night run incremental checks.

8. Password Changer: Configuration File

The *test file* implements the attempt to compromise between allowing the user to choose the password and having the system assign one by allowing a system administrator to limit the user's choice of passwords. This file contains a series of tests, each of which the proposed password must fail in order to be accepted as an allowed new password. The file also contains some control information which dictates how certain parts of the tests are done.

Horizontal tabs (ASCII HT) are important because they serve as separators on the lines. In this section, they will be represented by the symbol <HT>.

8.1. Comment Lines

These are lines that start with ''#'', and are ignored. Note that ''#'' must be in column 1 of the line; otherwise it will be interpreted as a (mangled) test and a syntax error will be reported.

8.2. Test Lines

These are lines which are interpreted as tests. No special format is used; rather, any line which does not fall into the types described in the remainder of this section is a test line.

The line has the format

test <HT> *error message*

The tests and error messages will be discussed separately.

8.2.1. Format of Tests

To be accepted as a password, a proposed password must undergo a series of tests. The tests have the following format:

```
test    ::= '(' test ')'
        |   test [ '&' | '|' ] test
        |   [ '!' | '~' ] test
        |   numexp numop numexp
        |   string strop string
numexp  ::= '(' NUMBER ')'
        |   NUMBER mathop NUMBER
        |   [ '+' | '-' ]* NUMBER
string  ::= '"' STRING '"'
        |   '[' FILE ']'
        |   '{' PROGRAM '}'
mathop  :=  '+' | '-' | '*' | '/' | '%'
numop   :=  '==' | '!=' | '>=' | '<=' | '<' | '>'
strop   :=  '==' | '!=' | '=~' | '!~'
```

where the primitives are defined as follows.

NUMBER This is a string of decimal digits. It is always interpreted as a decimal number.

STRING This is any sequence of ASCII characters. The standard C escapes are recognized:

Recognized Escape Sequences for Strings			
name	*ASCII name*	*octal code*	*escape*
backspace	BS	010	\b
horizontal tab	HT	011	\t
newline	NL (LF)	012	\n
form feed	FF (NP)	014	\f
carriage return	CR	015	\r
double quotes	"	042	\"
backslash	\	134	\\
bit pattern		*ddd*	*ddd*
anything else	*x*		*x*

FILE This is the name of a file that contains strings; the file is read, and each string is tested as indicated. Note that the format of the file is one string per line.

PROGRAM This is the name of a UNIX command that, when run, writes a set of strings on its standard output. The output is read, and each string is tested as indicated. Note that the format of the output is one string per line.

The operations and relations in the *mathop* and *numop* definition are the usual ones: the *mathop* operators are (in order) addition, subtraction, multiplication, division, and remainder; the *numop* relations are (in order) equal, unequal, greater than or equal to, less than or equal to, less than, and greater than. The relations in the *strop* definition allow both string comparison and pattern matching; they are (in order) equal, unequal, match, and nonmatch. The first two do character comparison of the strings; the latter two treat the right hand string as a pattern, and determine if the left hand string matches (or does not match) that pattern. The matching uses the pattern matcher provided by UNIX

It is an error (and will be reported as such) if both sides of a *strop* relation are not strings. That is, the output of a program cannot be compared to strings in a file, and vice versa, not can the contents of two files or the outputs of two programs be compared. This is not much of a restriction, because it can be done by writing a program that prints whether the test succeeded or failed, and then comparing the output with the expected output.

Numerical expressions, as summarized in the definition of *numexp*, are evaluated with the usual mathematical precedence; the symbols in the second line of that definition mean addition, subtraction, multiplication, division, and remainder, respectively. Each number may be preceded by a plus or minus sign, and parentheses are to be used for grouping.

Tests are logical; that is, a test is either *true* or *false*. The operators '&' and '|' allow logical disjunction (anding) and logical conjunction (oring) of tests, and the operators '!' and '~' both negate the result of the test. As with numerical expressions, parentheses are used for grouping.

If the test is *true* for the proposed password, the proposed password is rejected; otherwise, it is accepted.

8.2.2. Escapes

Before being evaluated, each test is scanned for specific character sequences, and these are replaced by data specific to the current execution of the password changing command. The notation used is very similar to that of *printf*(3). These *interpolation sequences* are divided into two classes, numbers and strings, and both types have the same general form:

%dm.nfc

where the boldface characters are literal and the other characters have the following meanings:

d If present, *d* is a '**–**'. If it is present and the quantity being interpolated is a string, it is reversed before being interpolated; if the quantity is a number, the number is subtracted from the number of significant characters in the password. (The single exception to this is the interpolation character **v**; for that character only, the value being interpolated is subtracted from 1.)

m.n If present, *m* and *n* are numbers. If the quantity being interpolated is a number, these fields are ignored. If a string, the characters from position *m* to position *n* inclusive are interpolated. If *n* is omitted or is larger than the length of the string, the characters from position *m* to the end of the string are used; if *m* is missing, characters from the beginning of the string to *n* are used; if *m* is larger than the length of the string, nothing is interpolated; and if *m* is present but both the period and *n* are missing, the first *m* characters of the string are interpolated. For example, if the string associated with the interpolation character **u** were **bishop**, "%u" would be replaced by **bishop**, "%2u" would be replaced by **bi**, "%2.u" would be replaced by **ishop**, "%2.4u" would be replaced by **ish**, and "%.4u" would be replaced by **bish**.

f For numbers, this is ignored; for strings, this indicates how the string is to be interpolated. The following shows possible values for this field and what they mean.

Format Control in Escapes	
control	*meaning*
^	alphabetic characters are made upper case
*	alphabetic characters are made lower case
|	if the first character is alphabetic it is capitalized
#	interpolate the length of the string; in this case the interpolated sequence is numeric

c This character says what is to be interpolated. The possible values (and whether they are numeric or string) are given in the following table. Any other character is printed as is.

Escapes		
character	*type*	*meaning*
A..Z	string	user-definable escape
a	number	number of alphanumeric characters in proposed password
b	number	number of alphabetic characters in proposed password
c	number	number of capital letters in proposed password
d	string	domain name of computer
f	string	first name of user
i	string	initials of user
l	number	number of lower case letters in proposed password
m	string	middle name of user
n	string	full name of user
o	string	office of user
p	string	proposed password
q	string	current password as entered to the prompt
s	string	surname of user
t	string	telephone number of user
u	string	login name of user
v	number	1 if proposed password has both upper and lower case alphabetic characters, 0 if not
w	number	number of digits in proposed password

For example, the sequence

```
%1f%1m%1s
```

would interpolate the first letter of the user's first, middle, and last name; if any of these is not defined, that sequence is ignored. Note that this is the same as the interpolation sequence "%i".

Now, suppose the user's new password is to be "SleeZe&1". Then "%a" is replaced by 7, "%b" is replaced by 6, "%c" is replaced by 2, "%l" is replaced by 4, "%v" is replaced by 1, and "%w" is replaced by 1.

8.2.3. Some Examples

Suppose passwords with 6 or fewer characters all being lower case letters are to be rejected. The following test evaluates *true* for all such proposed passwords and *false* for all others:

```
(%#p <= 6) & ("%p" =~ "[a-z]*")
```

As a second example, people often make their password the same as their login name, or their login name reversed, with variations in the case of the characters The following test will prevent this:

("%*p" == "%*u") | ("%*p" == "%-*u")

Note that the login name, its reverse, and the proposed password are all compared with alphabetic characters in lower case.

New California automobile license plates have a digit followed by three letters followed by three digits. To reject all proposed passwords which might be valid new California automobile license plates, use the following test:

("%*p" =~ "[0-9][a-z][a-z][a-z][0-9][0-9][0-9]")

Now, suppose you have a set of words which a specific user might use as his password; for example, suppose user *bishop*'s wife's name is " Holly", her maiden name is "Olson", and his children are named "Heidi" and "Steven". Any of these would be a possible password, and very easy for someone who knows that user to guess. In this case, put those passwords into a file with the user's name as part of the file name (for this example, call the file */etc/passwds.bad/bishop.str*). Then the test

"%p" == [/etc/passwds.bad/%u.str]

will reject any proposed password in the file.

An alternative would be to list all the words for all users in a file (call it */etc/passwds.bad*), and put the user's name followed by a colon and (optionally) some white space. So, in the above, the lines for *bishop* would be:

```
bishop:    Holly
bishop:    Olson
bishop:    Heidi
bishop:    Steven
```

and the test

"%p" == { grep '^%u:[\t]*' /etc/passwds.bad | sed -n 's/[^:]*:[\t]*\(.*\)/\1/p' }

would reject any proposed password deemed unsuitable for that user.

The standard UNIX *passwd*(1) program advises users that "new passwords should be at least five characters long, if they combine upper and lower-case characters, or at least six characters long if in monocase." To force all passwords to meet this requirement, use the test

((%#p >= 5) & (%v == 1)) | ((%#p >= 6) & (%v == 0))

8.2.4. Error Messages

As stated before, the test is terminated by a tab or a newline. If a tab, the remainder of the line is treated as an error message. When a user's proposed password satisfies a test, by default the error message

password invalid — no change

is printed. This is unhelpful, to say the least. Users should be told why their specific proposed password was not acceptable. To cause a specific error message to be printed when a password satisfies a given test, put the error message on the same line as the test and separate it wit a horizontal tab character. For example, if a user asked that the

password be set to the user's login name, the line

"%p" == "%u" <HT> password cannot be your login name

would cause the proposed password to be rejected with the error message

password cannot be your login name

8.3. Interpretation of the GECOS Field

The *gecos* field contains information about the user, usually his or her name, office, and telephone number. This information is used to set the values of various interpolation characters.

Basically, the format used at the site is defined by a *scanf*(3) format string followed by a list of interpolation characters to which those strings are to be assigned. For example, suppose a site kept the name, the office number, and the telephone number of users in its *gecos* field; a typical field might be "Matt Bishop,N230-102,6921". The line that would set the interpolation characters properly would be

GECOS: "%s %s,%s,%s" f s o t

and after processing this line, "f" would be set to the string "Matt", "o" would be set to the string "N230-102", "s" would be set to the string "Bishop", and "t" would be set to the string "6921".

Another, better, way to do this is to use the line

GECOS: "%[^,],%[^,],%s" n o t

This would set the interpolation sequence %n to the full name ("Matt Bishop" and %o to the office name regardless of any blanks in either, and the first and last names would be derived from the full name.

Only the interpolation characters A to Z, f, i, m, n, o, s, and t may be set using this method: The *gecos* field is scanned *after* all occurrances of the character "&" in it have been replaced by the user's login name. Also, note that if the *scanf* fails (for example, if the *gecos* field above had no commas), none of the values of the interpolation characters are reset. So, if the password file has *gecos* fields using a variety of formats, a series of these lines should be set up; as the lines are processed, those that match set (or reset) the interpolation characters listed.

If the password changing program is compiled with CHFN set, two variants of this line come into play. The line

SETGECOS: "%s %s,%s,%s" f s o t

acts just like the line beginning with "GECOS:", except that when running to change the *gecos* field, the user will be prompted to change each field. In the above, the dialogue between the user and the program would be:

First Name [Matt]: *John*
Last Name [Bishop]: *Doe*
Office [N230-102]: *310 Bradley Hall*
Phone Number [6921]: *2415*

and the new *gecos* field would be "John Doe,310 Bradley Hall,2415". Had the line been

SETGECOS: "%[^,],%[^,],%s" n o t

the dialogue between the user and the program would have been:

Name [Matt Bishop]: *John Doe*
Office [N230-102]: *310 Bradley Hall*
Phone Number [6921]: *2415*

and the *gecos* field would have been as above. Note this will occur for the first line beginning with "SETGECOS:" that matches the pattern; all following lines will be ignored.

The second variant comes into play when none of the "SETGECOS:" lines matches the format of the *gecos* field. This line takes the following form:

FORCEGECOS: "%s %s,%s,%s" f s o t

It is completely ignored unless the user is resetting his *gecos* field; then, it prompts for the data requested by the interpolation characters and writes it out in the indicated format. For obvious reasons, there should be only one such line in the file, and it should follow all "GECOS:" and "SETGECOS" lines.

As an example, suppose the user is changing his or her *gecos* field, and the configuration file containes the following lines:

GECOS:	"%s %s" f s
GECOS:	"%[^,],%[^,],%s" n o t
SETGECOS:	"%s %s,%s,%s" f s o t
GECOS:	"%s %s %s" f m s
FORCEGECOS:	"%s,%s,%s" n o t

If the *gecos* field is "Mary Smith", the first line will set f to "Mary" and s to "Smith". None of the other lines match this format, so the "FORCEGECOS:" line will cause the program to question Mary as follows:

Name [Mary Smith]: *Maria N. Smith*
Office []: *31 Dwinelle Hall*
Phone Number []: *1221*

Now suppose the *gecos* field were "Mary Smith,21 Cory,1234". In this case, the second line would set n to "Mary Smith", o to "21 Cory", and t to "1234". But the third line also matches the *gecos* field, and since it is a "SETGECOS:" line the dialogue would go:

First name [Mary]: *Maria*
Last name [Smith]: *Smith*
Office [21 Cory]: *31 Dwinelle Hall*
Phone Number [1234]: *1221*

The "FORCEGECOS:" line would not be used since a "SETGECOS:" line matched the format of the *gecos* field.

To change the prompt for any of these statements, use a line of the form

PROMPT: s "Surname [%s]: "

and then whenever a value for the interpolation character is needed, the prompt

Surname [*last name here*]:

would be printed.

8.4. Setting Variables Unconditionally

Sometimes it is desirable to set variables independently of information in the password file. For example, suppose a site keeps the names of computer users and their spouses in the file "/usr/adm/spouses". There is no way to use the above mechanism to access these names, but a site administrator should be able to use this data for testing passwords. This may be done by the "SETVAR" statement, which has the form

SETVAR: *character value*

Character is any of the user-definable interpolation characters A to Z, f, i, m, n, o, s, and t; *value* is the value to be assigned to it. *Value* may be a string, a file name, or a program. Only the first line of a file, or of the output of a program, will be used. All C escapes are recognized, and the syntax is the same as for the tests. So,

SETVAR: W { grep '%u' /usr/adm/spouses | sed -n 's/%u[\t]*//p' }

would assign to W the name of the spouse of the user,

8.5. Number of Significant Characters in the Password

By default, all pattern matching and string comparisons use only the first 8 characters of the password. To change this, put the line

SIGCHARS: *number of significant characters*

before any tests. For example,

SIGCHARS: 6

instructs the program to consider only the first 6 characters of the password. If the number of significant characters is 0, all chaacters are considered significant.

Suppose there are *n* significant characters. For all purposes, the password acts as though it was exactly *n* characters long. String comparisons are done on the first *n* characters only; if either string is longer than *n* characters, the excess in both is ignored. (For example, assuming 8 significant characters, if a user picked the password "ambidext123", and the passwords were tested against dictionary words, the word "ambidextrous" would match the proposed password, since the first 8 letters of each are the same.) Pattern matching is done on a password of length *n*; extra characters in the password (but *not* in the pattern) are ignored. (In the above example, suppose the proposed password "ambidext123" were being compared against the pattern "[aA][mM][bB][iI]dextrous". The pattern match would fail, since "ambidext" does not match the pattern. But if the pattern were "[aA][mM][bB][iI]dext", the pattern match would succeed.) The idea here is to treat the proposed password exactly as UNIX would treat a regular password.

On most UNIX systems, there may not be more than 8 significant characters in the password, because the password encryption function only uses the first 8 characters. In fact, the standard password reading subroutine only returns up to 8 characters, and discards the rest. But beware: if the number of significant characters is set to more than 8, things will not work as expected, because even though there will be 8 characters in the password, the strings used in comparisons may have more than 8 characters. So, continuing the above example, if the number of significant characters were set to 9, the proposed password "ambidextrous" would not match the dictionary word "ambidextrous"; the comparison function would compare the proposed password "ambidext" with the first 9 characters of "ambidextrous". Since there is an "r" at the end of the dictionary word when truncated, the string comparison fails.

8.6. Logging Errors and Debugging Output

The password changer has the ability to log various intermediate results, syntax errors, problems, and so forth. It should be emphasized thet *the logging mechanism will never log a proposed password unless it is being rejected;* this preserves the integrity of the user's password, but allows experimental or debugging data to be gathered without compromising accounts.

Logging is controlled by lines in the configuration file of the form

LOGTYPE: *type,type,...* <HT> *location*

The *type* s of logging that may be done are:

system — This logs system errors (such as files which cannot be opened); these errors may be reported to the user, or they may simply cause the password program to print a more general message saying that the password cannot be changed now.

use — This logs who is running the program, whose password is being changed, the name of the password file in which the change is being made, and the name of the configuration file being used.

result — This logs the result (success or failure) of the attempt to change the password.

item — If the attempt to change the password fails because the proposed password satisfies a test, this logs the line number of the specific test that was satisfied and the error message printed for the user.

syntax — This logs any syntax errors in "GECOS:", "LOGTYPE:", or test lines. If a test contains a syntax error, the test cannot be satisfied.

debug — This logs debugging information. It prints all parameters set by "GECOS:" and "SIGCHARS:" lines, and each test with all escapes fully expanded as well as the result of the test.

all — This turns on all types of logging.

clear — This terminates the logging to the particular *location*.

As stated above, when logging many types of information, the names are listed with commas between them. The types are evaluated in the given order, with successive types being or'ed in. Initially, no logging is done. To turn off specific types of logging, put an

exclamation point '!' before the type. For example,

LOGTYPE: clear,all,!system <HT> *location*

turns on all types of logging except system logging. Note that

LOGTYPE: all
LOGTYPE: use,result,syntax,item,debug <HT> *location*

would not do the same thing, since the first line turns system logging on, and the types on the next line are just or'ed in.

There are three or four ways to deal with logging information:

| *command* The logging output is used as input to the program *command*. This is especially useful when *command* is a command to mail the output somewhere.

> *file* The logging output is appended to the file *file*. Note it does not overwrite the contents of that file.

syslog This option is viable only if your system supports a system logging function. It routes logging output to the system logging function. If this option is not viable, it acts like the next location.

stderr This writes logging output to the standard error (usually the user's screen.) By default, logging output goes to the file ''/etc/passwd.log''.

For example,

LOGTYPE: syntax,system <HT> | mail staff

sends all system and syntax error messages to all members of the *staff* mail alias.

8.7. Sample Configuration File

In this section, we shall examine a typical configuration file. The file is printed in a fixed-width, smaller font; everything else is commentary.

```
# Sample Password Configuration File
#
# Matt Bishop, Oct. 6, 1987
#
```

These are the header commentary, to say what the file is.

```
#
# establish GECOS format and define some variables
# note we allow users to reset the GECOS field, but
# want them to conform to the format "name,office,telephone",
# so we use the FORCEGECOS line to do this
#
GECOS:          "%s" s
GECOS:          "%s %s" f s
GECOS:          "%s %s %s" f m s
GECOS:          "%s %s %s %s" f m s P
GECOS:          "%[^,],%[^,],%s" n o t
FORCEGECOS:     "%s,%s,%s" n o t
```

At this site, two types of *gecos* entries occur: those of the form "*name*", and those of the form "*name,office,phone*". The first four try to match the first form (assigning the interpolation characters appropriately), and the last tries to match the second form. If the user is simply changing the *gecos* field, regardless of which form is matched, the new entry will be made to match the second form.

```
#
# for system administration, we log any problems
#
LOGLEVEL:           clear,system,syntax        | mail pwlist
#
# for experimental data, we record use, result, and item;
# note a valid password is never logged
#
LOGLEVEL:           use,reult,item             > /usr/adm/passwd.expdata
```

We are logging two types of data. Whenever an error occurs, either in the configuration file or due to the system's being unable to access something, the appropriate error message is mailed to the address *pwlist*. Whenever someone tries to change a password, that attempt and its success or failure are logged in the file "/usr/adm/passwd.expdata". If the attempt fails, the proposed password and the reason it was rejected are also logged.

```
#
#=============================================
# general tests
#
%#p<6                  password must be at least 6 chars long
%#b>1&%#v=0            alphabetic chars, must be mixed case
```

Here, the minimum requirements for a password are checked: it must be at least 6 characters long, and if there is more than one alphabetic character, both upper and lower case alphabetic characters must occur.

```
#
# some checks on lazy people
#
"%*p"=="%*u"        login name not allowed as password
"%*p"=="%-*u"       reversed login name not allowed as password
"%*p"=="%*f"        first name not allowed as password
"%*p"=="%-*f"       reversed first name not allowed as password
"%*p"=="%*i"        initials not allowed as password
"%*p"=="%-*i"       reversed initials not allowed as password
"%*p"=="%*s"        last name not allowed as password
"%*p"=="%-*s"       reversed last name not allowed as password
"%*p"=="%*o"        office not allowed as password
"%*p"=="%-*o"       reversed office not allowed as password
"%*p"=="%*t"        phone number not allowed as password
"%*p"=="%-*t"       reversed phone number not allowed as password
```

These tests eliminate some obvious passwords. Note we map everything into lower case first; that way, we need not worry about people using their login name with the first letter capitalized, for example.

```
#
# knock off (most) licence plates
#
%w==%#p                                          NH license plate not allowed as password
((%w+1)==%#p)&&("%-1*p"=~"[a-z]"))     same
"%*p"=~"[0-9a-z][0-9a-z][0-9]*"          VT license plate not allowed as password
```

Here, we try to prevent users from using their license plate. We can't cover every case because most states will allow you to have any set of 8 (or so) characters on your license plate, but we can cover most. New Hampshire licence plates tend to be all numbers or all numbers with a trailing letter; the first two tests take care of them. Vermont license plates have two letters or numbers followed by a string of numbers; the third test eliminates them.

```
#
# site/host names
#
"%*p"=~"%h"              host name not allowed as password
"%*p"=~"%-h"             reversed host name not allowed as password
"%*p"=~"%d"              domain name not allowed as password
"%*p"=~"%-d"             reversed domain name not allowed as password
"%*p"=~"%h.%d"       domained host name not allowed as password
"%*p"=~[/etc/hosts.equiv]     trusted host name not allowed as password
```

Very often, users will make part of their password depend on the host name, the domain name, or the name of a trusted host. These tests reject any password with those as part. Note that we do not have any one-letter hosts here; if we did, we would have required exact matches.

```
#
# dictionary words -- look for strange capitalizations too
#
{tr A-Z a-z < /usr/dict/words} == "%*p"   password is in dictionary
```

This test compares the password to all words in the dictionary; comparison is done in lower case only. That way, capitalizing a word in the dictionary will not make it an acceptable password. Note this is done by running the command *tr* (1) on the dictionary.

9. Password Checker: Error Messages and What to Do

This section lists possible error messages from the password checking system and how to handle them. System errors may come from a number of other programs, and these are not included.

runcheck: unknown option *option*

The program **runcheck** was run with incorrect options. Check the manual (or section 6 of this document) and try again.

runcheck: –a flag requires addresses to send output

Runcheck was instructed to reroute output to someone other than the default administrators but no other address was provided. Rerun the program, giving the

appropriate address after the –a option.

runcheck: –d flag requires dictionary name

The flag indicating the name of a dictionary was given, but no dictionary was named. Rerun **runcheck**, naming the dictionary after the –d option.

runcheck: must check all passwords if not updating password files

The –n option was given and the –f option was not. Since **runcheck** has no way of determining which passwords were changed without updating the password files, it printsthis warning and sets the –f option automatically.

runcheck: WARNING: the system dictionary *filename* is empty.
This means that only illegal or empty passwords will
be found. Please check that you are issuing the runcheck
command properly.

The system dictionary constructed from the word lists, command line dictionaries, and password files (as appropriate) contains no words. The program will now only report missing passwords and (if the –i option is given) illegal passwords. This is not an error, but a warning because it probably is not what was intended.

testpwds: no more room (ualloc (malloc))
testpwds: no more room (ualloc (realloc))
testpwds: no more room (ialloc)
testpwds: no more room (stralloc)
testpwds: ran out of room (p2calloc)
testpwds: no more room (dalloc (malloc))
testpwds: no more room (dalloc (realloc))

These messages mean that too many passwords are being checked. Check fewer passwords.

testpwds: warning: mock password file is *filename*
testpwds: warning: dictionary file is *filename*

The password checker could not save the checkpoint files where it was told to, so it invented its own names. All future checkpointing done by the process is done to the named files.

day: localtime() returned NULL!

The system could not generate the expected name for checkpointing files. This should never happen, since it indicates serious system problems.

Usage: pwgrep strfile [file1 ...]

The program *pwgrep* was called incorrectly. Since it is only called by the script **runcheck**, this should never happen.

pwgrep: malloc: ran out of room [strsave]

pwgrep: malloc: ran out of room [psave]
pwgrep: realloc: ran out of room [psave]

These messages mean that the password files being checked contain too many entries. Try splitting the password files half.

subst: too many fields (limit 10)

Too many fields are being produced as input to **subst**. As this program is only called by the script **runcheck**, this should never happen.

getpw: unknown option *option*
diffpw: unknown option *option*
diffpw: too many options

The programs **diffpw** or **getpw** were called incorrectly. As these programs are only called by the script **runcheck**, this should never happen.

9.1. Troubleshooting

Runcheck has two debugging options: –v and –x. These act just line the options –v and –x to the Bourne shell *sh* (1) The –x option prints the commands and their arguments as they are executed; it is very useful when strange error messages occur, because it is propogated to all shell scripts invoked by **runcheck**, and so can be used to see precisely what command is giving the error. The –v option is useful when the scripts are changed; other than that, it's not much good.

Both options generate lots of output, and unless you know the Bourne shell quite well, you may find the output confusing. So use these options judiciously.

NAME

cap – capitalize the first letter of each line

SYNOPSIS

cap

DESCRIPTION

Cap copies the standard input to the standard output, and if the first character on the line is a letter, *cap* changes its case.

SEE ALSO

tr(1)

NAME

canexec – determine if a command can be executed

SYNOPSIS

canexec *command*

DESCRIPTION

Canexec attempts to execute the given shell script *command*; it is assumed to be a command script for the Bourne shell *sh*(1). If it succeeds, the exit status returned is that of *command*; otherwise, it returns an exit status of 1.

SEE ALSO

test(1)

NAME

cryptdes – encode/decode using the Data Encryption Standard

SYNOPSIS

cryptdes [**–i**] [password]

DESCRIPTION

Cryptdes reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *cryptdes* demands a key from the terminal and turns off printing while the key is being typed in. *Cryptdes* encrypts and decrypts with the same key:

```
cryptdes key <clear >cypher
cryptdes –i key <cypher | pr
```

will print the clear. (If the **–i** switch is present, *cryptdes* decrypts the input; if that switch is not present, it encrypts the input.)

Des implements the Data Encryption Standard proposed by FIPS. This standard is considered very secure. The only method of decrypting known now is a direct search of the key space, which is considered computationally infeasible.

Since the key is an argument to the *cryptdes* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *cryptdes* takes care to destroy any record of the key immediately upon entry. No doubt the choice of keys and key security are the most vulnerable aspect of *cryptdes*.

FILES

/dev/tty for typed key

SEE ALSO

Data Encryption Standard, Federal Information Processing Standard #46, National Bureau of Standards, U.S. Department of Commerce (January 1977)
A Fast Version of the DES and A Password Encryption Algorithm, Matt Bishop, Research Institute for Advanced Computer Science
crypt(1), des(3)

BUGS

There is a controversy raging over whether the DES will still be secure in a few years. The advent of special-purpose hardware could reduce the cost of a direct search of the key space enough so that such an attack is no longer computationally feasible.

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, the user assumes full responsibility for their use. Further, the author assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

AUTHOR

Matt Bishop
Research Institute for Advanced Computer Science
Mail Stop 230-5
NASA Ames Research Center
Moffett Field, CA 94035

Electronic mail addresses:
ARPA: mab@riacs.edu, mab@icarus.riacs.edu
UUCP: decvax!decwrl!riacs!mab, ihnp4!ames!riacs!mab, ucbvax!ames!riacs!mab

NAME

day – print the date in *mmddyy* format

SYNOPSIS

day

DESCRIPTION

Day prints the date as six digits, the first two being the number of the month (1–12), the next two being the number of the day of the month (1–31), and the last two being the last two digits of the year.

SEE ALSO

date(1)

NAME

diffpw – print the password file entries which differ

SYNOPSIS

diffpw[**−x**] [**−v**] *oldpwfile newpwfile*

DESCRIPTION

Diffpw compares the first two fields of each line in *oldpwfile* to those of each line in *newpwfile*, both of which are files in the format of *passwd*(5), and prints those lines in newpwfile which are not in, or differ from those in, *oldpwfile* . If any such lines are found, the exit status is 0; otherwise it is 1.

Options are:

−v,**−x** These are used for debugging; **−v** prints each input line as it is read, and **−x** prints each command and its arguments before it is executed. See *sh*(1) for more details.

FILES

/usr/local/adm/passwd/Misc/pwd*nnnnnc* first two fields of the password files

SEE ALSO

passwd(5)

NAME

getpw – fetch and update remote password file

SYNOPSIS

getpw[**−v**] [**−x**] *localfile pwfile*

DESCRIPTION

Getpw copies the password file *pwfile* to *localfile*. Both file names are of the form *host:filename*; if *filename* is omitted, it is replaced by "/etc/passwd" in *pwfile* and "passwd" in *localfile*. If *host:* is omitted, the local host is used.

Options are:

−v, −x

These are debugging options: **−v** prints each input line as it is read, and **−x** prints each command and its arguments before it is executed. See *sh*(1) for more details.

NAME

chfn, chsh, passwd − change password file information

SYNOPSIS

passwd [**−F** *file*] [**−t** *file*] [**−f**] [**−s**] [name]
chfn
chsh

DESCRIPTION

This command changes (or installs) the password, the GECOS information (using the **−f** option), or the login shell (using the **−s** option) associated with the user *name* (your own name by default).

When altering a password, the program prompts for the current password and then for the new one. The caller must supply both. The new password must be typed twice to forestall mistakes.

New passwords are tested to ensure that they are not easy to guess. The specific tests used vary from site to site. Unlike the standard password command, these rules may be abrogated only by changing the tests. The tests are contained in the file */etc/passwd.test*; the superuser may specify another test file with the **−t** option. This is intended for debugging a new test file.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

When altering a login shell, *passwd* displays the current login shell and then prompts for the new one. The new login shell must be one of the approved shells listed in */etc/shells* unless you are the super-user. If */etc/shells* does not exist, the only shells that may be specified are */bin/sh* and */bin/csh*.

The super-user may change anyone's login shell; normal users may only change their own login shell.

When altering the GECOS information field, *passwd* displays the current information, broken into fields, as interpreted by the *finger*(1) program, among others, and prompts for new values. These fields vary from site to site. Included in each prompt is a default value, which is enclosed between brackets. The default value is accepted simply by typing a carriage return. To enter a blank field, the word "none" may be typed. Below is a sample run for a format that specifies full name, office number, and telephone number:

Name [Johnathan D. Doe]:
Office [1600 Brilliant]: 212 Silly
Phone [(123)555-1212x2307]: none

Passwd allows phone numbers to be entered with or without hyphens. It is a good idea to run *finger* after changing the GECOS information to make sure everything is setup properly.

The super-user may change anyone's GECOS information; normal users may only change their own.

All changes are made to the file */etc/passwd* unless an alternate password file is specified with the **−F** option.

FILES

/etc/passwd The file containing all of this information
/etc/passwd.test The set of tests for new passwords
/etc/shells The list of approved shells

SEE ALSO

login(1), finger(1), passwd(5), crypt(3)
Robert Morris and Ken Thompson, *UNIX password security*

NAME

pwgrep – print the lines containing password entries

SYNOPSIS

pwgrep *pwlist pwfile*

DESCRIPTION

Pwlist contains a list of names or a list of names and encrypted passwords separated by a colon. *Pwgrep* prints those lines in *pwfile* which begin with that string. Unlike *grep* (1), it deals with strings and not patterns; unlike *fgrep (1)*, it forces matches to start at the beginning of a line.

SEE ALSO

fgrep(1), grep(1)

NAME

rev – reverse the line in a file

SYNOPSIS

rev [file] ...

DESCRIPTION

Rev copies each named file to the output, reversing the lines. If no files are named, input is taken from the standard input.

NAME

runcheck − run the password checker

SYNOPSIS

runcheck [−a*admin1,...*] [−d*file*] [**−f**] [**−i**] [**−n**] [**−p**] [−r*restart*] [**−u**] [**−v**] [**−x**]

DESCRIPTION

Runcheck attempts to find passwords by guessing words from a dictionary. It warns when it finds accounts with no passwords and accounts with passwords that it can guess. Normally, it determines which passwords (if any) have changed since the last check, and tests only those; this is called *incremental testing*.

Options are:

−a*admin1,...*
: Copies of lists of accounts with no passwords, accounts with invalid passwords (which therefore cannot be logged into), and accounts with passwords that *testpwds*(1) can guess are sent to the mailing addresses *admin1,...*.

−d*file* This treats *file* as a dictionary; the words in it are tried as passwords without further processing. Each line contains one word.

−f This forces all hosts to be checked whether or not the password files have changed since the last check.

−i This causes invalid passwords to be reported as well as empty or guessable ones. These passwords prevent anyone from logging in in a situation where a password is required, such as by using *login*(1); however, if the password mechanism can be circumvented (as with *rlogin*(1)), the account may be used.

−n With this option, no updating is done; the local copies of the password files are checked. Without this option, *runcheck* would update these first, and (if no changes, report that nothing had changed.

−p Normally, *runcheck* augments the system dictionary with words from the host's password file. To prevent this and force only the words in the system dictionary to be used, set this option.

−r*restart*
: If a run is terminated prematurely, it is checkpointed in two files /usr/local/adm/passwd/Misc/*mmddyy*.dic and /usr/local/adm/passwd/Misc/*mmddyy*.pwd, where *mmddyy* is the month, day, and year of the interrupted run. To continue this run, give this option with the name of the checkpoint file.

−u Notify users whose passwords have been guessed or who have no password.

−v, **−x**
: These are debugging options: **−v** prints each input line as it is read, and **−x** prints each command and its arguments before it is executed. See *sh*(1) for more details.

/usr/local/adm/passwd/Misc/nocheck list of users not to check
/usr/local/adm/passwd/Misc/nonotify list of users not to notify
/usr/local/adm/passwd/Misc/machines list of hosts to check
/usr/local/adm/passwd/Misc/dictions list of word lists to make dictionary
/usr/local/adm/passwd/Misc/pwd.found command to mail password found message
/usr/local/adm/passwd/Misc/pwd.nonecommand to mail empty password message
/usr/local/adm/passwd/Misc/pwd.illegal command to mail invalid password message
/usr/local/adm/passwd/Dict/sysdict composite of all dictionaries and password files
/usr/local/adm/passwd/Misc/exec.pwds program to make dictionary from password file

/usr/local/adm/passwd/Misc/exec.list program to make dictionary from word list
/usr/local/adm/passwd/Misc/exec.dict program to make dictionary from dictionary

SEE ALSO

Installing the Fast DES and UNIX Password Package, Matt Bishop
passwd(1), testpwds(1)

NAME

subst – substitute lines in a file

SYNOPSIS

subst *file*

DESCRIPTION

Input to *subst* consists of a series of lines divided into fields by horizontal tabs. For each input line, *file* is read, all character strings of the form "%*n*" (with *n* between 1 and 9 inclusive) are replaced by the *n th* field in the line, and the result is printed on the standard output. If there are fewer than 9 fields in the input line, the undefined fields are treated as empty. If "%" is followed by any other character, it is eaten; for example, to print a percent, use two percent signs.

SEE ALSO

sed(1)

NAME

testpwds – look for obvious passwords

SYNOPSIS

testpwds [**−c***cfile*] [**−d**] [**−l***lfile*] [**−m***levels*] [**−o***ofile*] [**−p***pfile*] [**−r***rfile*] [**−t***interval*] [**−v**] [*dictionary_file*]

DESCRIPTION

Testpwds attempts to find passwords by guessing words from one or more dictionaries. It warns when it finds accounts with no passwords, accounts with invalid passwords (which therefore cannot be logged into), and accounts with passwords that it can guess. By default, the syatem's password file is used, the potential passwords are read from the named *dictionary_files* or (if none are given) the standard input, and output is sent to the standard output. Dictionaries have the form of one word per line, and the words are tested in order.

Available options are:

−d Produce debugging output; this is useful only to maintainers of the program, and is available only if compiled in.

−l*lfile* When *testpwds* has completed its run, write a "mock password file" named *lfile* containing the names and encrypted passwords of all users whose passwords were **not** found. This file has the same format as a regular password file, but only the first two fields are non-empty; This file may be used as an argument to the **−p** option to test against another dictionary.

−m*levels*

This tells the program what types of password problems to report. Known levels are: **e** for empty password fields (ie, no passwords needed to log in); **i** for illegal passwords (ie, passwords with no corresponding cleartext); and **f** for found passwords (ie, passwords which have been cracked). More than one level may be specified in which case the union of all levels is printed. The default level setting is **ef**, since these indicate accounts that may be accessed illicitly.

−o*ofile* Append all output to the file *ofile* rather than to the standard output.

−p*pfile*

Check passwords in the file *pfile* rather than in the system password file. *Pfile* is assumed to have the format described in *passwd*(5). Multiple **−p** arguments may be given, but since all password files are read before any testing occurs, the order in which the passwords are checked is not necessarily the order in which the users are listed in the password files.

−v Enter verbose mode. In this mode, if a user's password cannot be found, a message is printed on the output. Normally, only accounts with no passwords, with illegal encrypted passwords, and with known passwords are printed.

CHECKPOINTING

Because running this program may consume quite some time, *testpwds* will checkpoint itself when signalled with any signal other than one used for job control (in 4.2 BSD, these are **SIGSTOP**, **SIGTSTP**, **SIGCONT**, and **SIGCHLD**; in System V, this is **SIGCLD**; see *signal*(2) or *signal*(3)). The checkpoint consists of two files: a "mock password file" (see the description of the −l option, above) and a "dictionary list file," which contains information about the dictionaries being used. By default, the first of these is named "pwds*xxxxxx*.pwd" and the second is named "pwds*xxxxxx*.dic" *Testpwds* may be restarted at a later date from the checkpoint by specifying "pwds*xxxxxx*.pwd" as the password file using the **−p** option, and "pwds*xxxxxx*.dic" as the restart file (using the **−r** option described below). Any number of restart files and mock password files

may be named; the restart files will be processed in order.

Options are:

−c*cfile* Set the name of the checkpoint files to *cfile*. The mock password file will be named "*cfile*.pwd" and the dictionary list file will be named "cfile.*dic*"

−t*interval*
Checkpoint the run every *interval* seconds. The minimum acceptable value for *interval* is 300 (that is, 5 minutes.) If this option is omitted, checkpointing is done only when one of the above-named signals is received.

−r*rfile* Restart an interrupted run using *rfile* as the restart file.

If **SIGHUP** is issued, the run continues after the checkpointing. If any other signal, except for the job control signals, is received the run will be checkpointed and then terminate.

FILES

/etc/passwd default password file

SEE ALSO

"An Application of a Fast Data Encryption Standard Implementation," Matt Bishop
passwd(1), crypt(3), des(3), passwd(5)

AUTHOR

This program was inspired by one that originated at the School of Electrical Engineering at Purdue University.

NAME

des – fast implementation of the Data Encryption Standard

SYNOPSIS

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char key[64];

encrypt(block, flag)
char block[64]

des_schedule(inverse)
int inverse;

des_key(bits)
char key[8];

des_run(mesg)
char mesg[8];

#include des.h
Unit *shc_crypt(key, salt)
char *key, *salt;

#include des.h
Unit *decrypt(password)
char *password

int des_bits(), cry_bits(), shc_bits()
int des_path(), cry_path(), shc_path()
int des_permkey(), cry_permkey(), shc_permkey()
```

DESCRIPTION

These routines implement the DES as described in the standard, but in a way that is much faster (typically, on the order of 25 times faster) than the standard routines. *Crypt*, *setkey*, and *encrypt* provide the standard interface to these routines (see *crypt*(3)). The other entry points provide a faster interface.

PASSWORD ENCRYPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is normally a user's typed password. The second is a two character string chosen from the set [./0-9A-Za-z]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt a constant string repeatedly. The returned value points to the encrypted password, which uses characters from the same set as the salt. The first two characters are the salt itself.

UNIX INTERFACE TO THE DES ROUTINES

Setkey sets the key of the DES algorithm. The key should be decomposed so each character element of the array *key* contains one bit of the key in the lowest order bit. *Encrypt* does the encryption or decryption. The first argument to it is likewise the message to be encrypted or decrypted, with one bit per character element. This array is modified in place to a similar array representing the bits of the input after having been subjected to the DES algorithm using the key set by *setkey*. If the second argument *flag* is 0, *mesg* is encrypted; if not, it is decrypted.

DIRECT ACCESS TO THE DES ROUTINES

Des_schedule determines the order of the key schedule; if its argument is 0, the order generated is suitable for encryption, and if the argument is 1, the order is suitable for decryption. *Des_schedule* must be called before *des_key*. *Des_key* generates the appropriate key schedule for the translation algorithm. Its argument is the key, which is an array of up to eight characters. (Note that if the key is under eight characters, *all* characters after the final one in the key *must* be '\0'.) *Des_run* takes the eight character *mesg* and either encrypts or decrypts it (as indicated by *des_schedule*) using the key schedule generated by *des_key*. The transformation is done in place. Notice each block is eight characters long; do not drop any bits or the transformation will be uninvertible!

PASSWORD TESTING FUNCTIONS

The functions *shc_crypt* and *decrypt* are provided to allow quick testing of passwords. *Decrypt* takes the encrypted form of a password, partially decrypts it, and returns a pointer to that partial form. *Shc_crypt* takes a plaintext password and the associated salt and returns a partially encrypted form of the password. If the encrypted password was produced by encrypting the plaintext password and associated salt, the two partial forms will match. If not, the two partial forms will not match.

SUPPORT FUNCTIONS

The following functions provide information about the way the DES routines work.

Des_bits, *cry_bits*, and *shc_bits* return the number of bits that the *encrypt* (and *des_run*), *crypt*, and *shc_crypt* routines were designed to use (this is usually, though not always, the word size of the machine.)

Des_path, *cry_path*, and *shc_path* return the number of bits *encrypt* (and *des_run*), *crypt*, and *shc_crypt* use to access the S boxes within the innermost loop of the algorithm.

Des_permkey *cry_permkey*, and *shc_permkey* return 1 if *encrypt* (and *des_run*), *crypt*, and *shc_crypt* compute the key schedule using a single permutation, and 0 if not.

INTERMIXING CALLS

There are three key schedules used: one by *crypt* , one by the ordinary DES routines, and one by *shc_crypt* . Calls to these functions may be intermixed freely, since the key schedules are independent. However, calling *setkey* and *encrypt* sets a key schedule ordering that uses the same storage as the key schedule for *des_key* and *des_run*. It is therefore exceedingly unwise to mix calls to *setkey* and *encrypt* with calls to *des_key* and *des_run*.

WARNINGS

Crypt, *shc_crypt* , and *decrypt* all return pointers to (private) static storage areas, so calling the same function twice destroys the result of the first call unless it has been copied elsewhere.

FILES

/usr/local/lib/libdes.a library

SEE ALSO

Data Encryption Standard, Federal Information Processing Standard #46, National Bureau of Standards, U.S. Department of Commerce (January 1977)

A Fast Version of the DES and A Password Encryption Algorithm, Matt Bishop, Research Institute for Advanced Computer Science

AUTHOR

Matt Bishop
Research Institute for Advanced Computer Science
Mail Stop 230-5
NASA Ames Research Center

Moffett Field, CA 94035

Electronic mail addresses:
ARPA: mab@riacs.edu, mab@icarus.riacs.edu
UUCP: decvax!decwrl!riacs!mab, ihnp4!ames!riacs!mab, ucbvax!ames!riacs!mab