Anatomy of a Proactive Password Changer

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
6188 Bradley Hall
Hanover, NH 03755

1. Introduction

The issue of poor user selection of passwords has been discussed in many papers [5][6] and need not be repeated here. Among the techniques used to overcome these problems are random generation of passwords [2], challenge-response techniques [4], key crunching [3], and the examination of user-selected passwords either by cracking them or by analyzing them before allowing the password to be changed. In this paper we look at a program specifically designed to do the latter.

This paper will describe a new version of the UNIX password changing program called *passwd+*. This program provides extensions to both the password changing facility and the password checking facility. The former allows users to be given full responsibility for, and control over, accounts other than their own; the latter allows the system administrators to constrain password selection so that users cannot install passwords deemed easily guessable.

2. Password Changing

The standard UNIX system password changing program *passwd*(1) [7] allows users to change one of three types of information: their password, their login shell, and their user information (called the "GECOS information" here). If password aging is enabled, a user may be unable to change his or her password; however, any user may change his or her login shell or GECOS information at any time. Of course, the superuser may change any user's information at any time.

Different vendors have extended this program in ways appropriate to their environment and configuration; for example, some versions have an option to print out the time until the user's current password expires. These extensions must be allowed for in any replacement program.

In what follows, we describe the relevant components of *passwd+* from a system administrator's point of view. We explain why each feature is present, how it is used, and give several examples.

When *passwd+* starts, it obtains information about the user named on the command line (or the current user) from the password file. In what follows, this will be called the *current information*. All functions are performed upon it, and this information resides in memory until it is written back to the password file.

2.1. User Interfaces

Three interfaces allow the system administrator to replace the current *passwd*, *chsh*(1), and *chfn*(1) programs transparently to the user. The fourth interface provides an interactive editor for the password, login shell, and GECOS information. In addition, the user can list the current information, obtain help, change the user whose password file information is being edited, and obtain the type and name of the password file being edited.

The interactive interface begins by printing a "message of the day" and then places the user in an interactive mode which has the following commands:

`exit, xit`
  exit without saving changes made to the current information

The support of grant NAG 2-628 from the National Aeronautics and Space Administration to Dartmouth College is gratefully acknowledged. Portions of this work were done while the author was visiting the Department of Computer Science at the University of California at Davis, CA.

`finger`, `gecos`
: change the current GECOS information; the user is prompted as necessary.

`help` print a help message

`info` print a message naming the current password file, configuration file, and user:

```
Current user: bishop
Current password file: * system *
Current configuration file: ./Sample/Config
```

`list` print the current information in an easy-to-read format:

```
name          bishop
password hash 3zvhvQtnxs9r/
user id       77
group id      20
GECOS         Matt Bishop
home dir      /usr/windsor/bishop
login shell   /bin/csh
expires on    *turned off*
can change on *turned off*
```

`password`
: change the current password; the user is prompted as necessary

`quit` exit; if there are unsaved changes, this will give an error message

`shell` change the current login shell; the user is prompted as necessary

`write` save the current information in the password file.

Unknown arguments are passed to a system-dependent routine which can act accordingly. This allows extensions to be added on a per-system (and per-site, if appropriate) basis.

2.2. Configuration File

The actions of the password changing part of the program are controlled by a configuration file; this allows sites to alter the default behavior as desired without having to edit C code and/or recompile the program. Although we shall discuss the specific components of this file below, a few general comments are in order.

The configuration file is read once, when *passwd+* starts. Reading is done by lines, and if the line just read ends with a backslash, the next line is taken as a continuation and is appended. (Because the input is stored dynamically, lines may be as long as desired.) The line is examined to see if it applies to the current user (here, "current user" is the user with the real UID of the process). If not, it is discarded. Otherwise, it is processed.

Because some strings in the configuration file may have blanks or other separators in them, *passwd+* observes two escape conventions. The first, a backslash followed by a character, interpolates the character literally, unless that character is a newline. The second, a sequence of characters not containing an unescaped newline and surrounded by double quotation marks, is taken to be a single string. For example, the following two sequences are both read as "my shell" with a blank between the "y" and the "s":

```
my\ shell
"myshell"
```

These conventions are followed whenever any reading of the configuration file is done.

An unescaped sharp sign "#" begins a comment which extends to the end of the line.

2.3. Permissions

The ability to change information is controlled on a per-user basis by the configuration file. There are two types of controls: `validate` allows a user to change a set of fields after supplying his or her password, and `novalidate` allows the changes without the user supplying a password. For example, the lines to provide controls equivalent to those of the standard password changer are:

```
novalidate root :all:
validate :password: :self: :self:
novalidate :gecos: :shell: :self: :self:
```

The first line says that the superuser (`root`) can change any of the information fields for all users without supplying a password. The second line says that the user of the program (`:self:`) can change his or her password, but must supply his or her current password. The third line allows the user to change his or her login shell and GECOS information without supplying a password.

As an example of how this mechanism can be used, suppose a professor with account name *bishop* is to be allowed to change the password and GECOS information for any accounts issued to his class. If the class accounts are *cs5801*, *cs5802*, and *cs5803*, the following lines suffice:

```
validate :password: bishop cs5801 cs5802 cs5803
novalidate :gecos: bishop cs5801 cs5802 cs5803
```

Note that *bishop* must enter his password to change any of the passwords of the class accounts, but need not do so to change the GECOS information. Further, as no permission is given to change the login shell of the class accounts, only the students (or the superuser) can do that.

The first word following the `validate` or `novalidate` is the type of information to which the remainder of the line applies; if this is omitted, the remainder applies to all types. Legal values are:

`:shell:`
: login shell information

`:password:`
: password information

`:gecos:`
: GECOS information

(Incidentally, the semicolon was chosen as a delimiter because that character cannot be used in an account name.)

The next word is the user name; if it is not that of the current user, the line is ignored. If it matches that of the current user, the remainder of the line contains the account names for which the current user can change the previously-indicated type of information. In addition to account names, the following special names have special effects:

`:all:` means all accounts

`:none:`
: means no accounts; this is useful with accounts that no user should ever log into

`:default:`
: means all accounts for which the current user has not previously been given permissions

`:self:`
: the account belonging to the current user

Following the principle of fail-safe defaults, unless access is granted by a validate or novalidate control line, the request to change information is denied.

As a final example, the following control lines allow the standard changing, except that the accounts *ftp*, *uucp*, and *audit* may never change their own passwords:

```
novalidate root :all:
validate :password: :self: :self:
novalidate :gecos: :shell: :self: :self:
validate uucp :none:
novalidate ftp :none:
```

Either `validate` or `novalidate` could be used in the last two lines.

As a safety and debugging measure, a list of distinguished user identification numbers can be made privileged at compile time. Any member of this set can change any information for any user, regardless of the settings in the configuration file. As distributed, this list is empty. However, as the superuser can edit a password file directly, there seems little point in not adding the UID of 0 to this list.

2.4. Changing the Password

If a user is allowed to change his or her password (as controlled by the permissions described in the previous section), *passwd+* prompts him or her for the new password. It is then subjected to proactive analysis. If a subprogram is to be used, it is named in the control line

```
check analysis_program
```

This program must expect as (standard) input the following information, separated by newlines:

proposed new password
old (current) password if available; this line is blank if not
account name
current hashed password
GECOS information
login shell
home directory
user identification (UID)
primary group identification (GID)
time when password can next be changed (in seconds since the epoch)
time when password expires (in seconds since the epoch)

All output from the program is sent to *passwd+*'s standard output and standard error, except that any lines on the standard output that begin with `**LOG**` are entered into *passwd+*'s log file. The exit status code of the analysis program determines what *passwd+* does next. If that code is 0, the password is accepted as hard to guess. If the exit status code is 1, the password is rejected as easy to guess. If the analysis program returns 2 or 3, there was an error that prevented the analysis program from using all its test but the password was deemed hard to guess (code 2) or easy to guess (code 3) according to those tests completed. Any other exit status code is treated as a 3.

If the analysis program did not complete successfully, the control line

```
onerror action
```

controls what happens. The following settings for *action* cause the indicated response:

`reject`
reject the password (that is, exit status code 2 causes rejection)

`accept`
accept the password (that is, exit status codes 2, or 3 cause acceptance)

`default`
accept the password if the exit status code is 2 and reject if it is 3

`internal`
run *passwd+*'s internal tests and accept or reject based on their success

Unless an `onerror` line appears, an exit code other than 1 or 2 causes the internal tests to be used.

If the analysis program fails, or none is named, or it cannot be executed for any reason, a series of internal tests are executed:

- if the password is the login name or the login name reversed (regardless of capitalization), the password is rejected;
- if the password is under 6 characters, it is rejected;
- if the password has no nonalphanumeric characters, it is rejected.

While these tests are not very adequate, they are the minimum qualities a reasonable password should possess. Note that unlike most implementations of the standard password changing program, these **cannot** be relaxed or disabled without modifying the code.

2.5. Changing the GECOS Information

Sites store information in the GECOS field using very different formats; further, the type of information stored at each site is different. For example, at the Research Institute for Advanced Computer Science, the typical GECOS information was stored as a name, an office, and a telephone extension number:

```
Matt Bishop,N238-102,46124
```

But in one department at Dartmouth College, faculty members have only their names stored:

```
Matt Bishop
```

and in another department at the University of California at Davis, the system accounts have GECOS fields of one word and user accounts have the user name followed by office number, year of graduation (if any), phone number, home phone number, and faculty sponsor:

```
Matt Bishop,4413 Chem Annex,27324,,Karl Levitt
```

(The home telephone number is omitted here.) Given this variety even within sites, the password changer should parse the GECOS field, determine what format is being used, and either use the same one or, if preferred, supply a new format. For this reason, control lines are needed to pick out the components of the field, tie prompts to the (so the user can be asked to update the information) and to reformat the fields.

The mechanism chosen to parse the fields was the system pattern matcher (the obvious alternative, the formats used by the reading function *scanf*(3), was rejected as too cumbersome). Currently a public-domain implementation of the Berkeley (UNIX System Version 7) pattern matcher, and the source code of the GNU *emacs* pattern matcher, are provided. As distributed, the Berkeley pattern matcher is the default. The relevant control line is

```
pattern pattern_matcher
```

with *pattern_matcher* being `bsd` for the Berkeley version and `gnu` for the GNU emacs version.

The pattern matcher is used to assign portions of the GECOS field to variables. For example, consider the line from the RIACS file above. In the following control line

```
getgecos "^\([^,]*\),\([^,]*\),\(.*\)$" fullname office extension
```

the pattern matches the format of the RIACS GECOS field. The first part of the pattern, `Matt Bishop`, is assigned as the value of the variable *fullname*; the second part, `N238-102`, is assigned as the value of *office*, and the third, `46124`, as the value of *extension*. If the pattern does not match the GECOS field, the line is skipped. The first line with a match does the assignment, and the process stops.

Next, the output format must be selected. Output formats are defined in `setgecos` control lines; these use a *printf*(3) format rather than a pattern. An appropriate output format for the RIACS line would be

```
setgecos "%s,%s,%s" fullname office extension
```

But before output can be done, the user must be prompted to update the current information, so a label and default value must be associated with each variable. The `associate` control line does this. For example, reasonable `associate` control lines for RIACS would be:

```
associate fullname Name none
associate office "Office Number" none
associate extension Extension 46363
```

The variable name comes first, followed by the prompt and the default value. Note that if the prompt is more than one word, any intervening white space must be escaped. If the default is omitted, the word "none" is used.

When the `setgecos` line is reached, the user will be prompted for a new value of each variable in the `setgecos` list. The prompt is followed by a default value, which is the current value (if any) or the default value named in the `associate` line. So, in this example the prompts would look line:

```
Name [default "Matt Bishop"]:
Office Number {default "N238-102"]:
Extension [default "46124"]:
```

Note that if nothing is typed, the default named in the prompt is used. To supply a blank entry (that ism, the null string, the word "none" must be typed. This is to conform to the standard UNIX change GECOS utility interface.

As a more complex example, the following control lines handle the two distinct formats described for the University of California at Davis GECOS fields:

```
getgecos “^\([^,]*\),\([^,]*\),\([^,]*\),\([^,]*\),\(.*\)$” name room \
         wphone hphone sponsor
setgecos “%s,%s,%s,%s,%s” name room wphone hphone sponsor
getgecos “^\(.*\)$ name
setgecos %s name
associate name Name none
associate room Office none
associate wphone “Office phone number” 27004
associate hphone “Home phone number” 5551212
associate sponsor “Faculty sponsor” none
```

If Matt Bishop wished to change his office entry and home phone number, here is what the session would look like (the responses to the prompts are in oblique typeface):

```
Name [default “Matt Bishop”]:
Office [default “4413 Chem Annex”]: 4414 Chem Annex
Office phone number [default “27324”]:
Home phone number [default “5551212”]: 5551313
Faculty sponsor [default “Karl Levitt”]:
```

If the superuser wished to change his GECOS information, which is simply the string

```
Charlie the Root
```

the second `getgecos` line would be used (as the pattern on the first line does not match the current contents) and the output format would be that of the next `setgecos` field. In this case the session would look like:

```
Name [default “Charlie the Root”]: The Root of All Evil
```

because the variable *name*’s value is set to `Charlie the Root`.

2.6. Changing the Login Shell

These lines control which shells may be used; they do not control who may change a login shell (the permissions lines control that). A user may change his or her shell only to those shells allowed by controls in this section. For example, the control

```
getusershell
```

allows a user to use the system’s shells (as listed in “/etc/shells” or, if that file does not exist, either “/bin/sh” or “/bin/csh”) as login shells. The control

```
shell /usr/bin/special_shell
```

allows any user to use the file “/usr/bin/special_shell” as a login shell. If users are to be allowed to use any file as a login shell, the control

```
anyshell
```

should be listed; if this is to apply only to a set of users, the account names of those users may be listed after the anyshell. Finally, the control

```
ownshell bishop
```

means that *bishop*’s shell can only be a file he owns; if the name is omitted, that restriction applies to all users.

As an example, here is the configuration which acts in the same way as the standard UNIX password changing program, in which the superuser can use any shell, but users can only use their own programs or standard system shells:

```
getusershell
anyshell root
ownshell
```

Note that the shell need not be executable so that the user can disable his or her account if desired.

Whether or not this is a feature or a bug is a matter of taste; it is present because the standard shell changing programs allow it.

2.7. Running Subprograms

Subprograms may be run at three points. First, if a user requests help, the password checking routine may be executed with a special flag. Second, if logging is done, the configuration file can cause log messages to be passed as input to a subprogram. Third, when the password checker itself is run, it is run as a subprocess. Given that all of these will be run with the super-user's effective UID, care must be taken in spawning them; the threats to the system by setuid-to-root programs have been discussed elsewhere.

The *passwd+* program thoroughly sanitizes the subprogram's execution environment before running the subprogram by taking the following precautions:

1. the shell environment variables are deleted;
2. the **PATH** environment variable is reset to a safe state (as distributed, "/bin:/usr/bin:/usr/ucb:/etc");
3. the **SHELL** environment variable is reset to a known default (as distributed, "/bin/sh");
4. the **IFS** environment variable is reset to the default value; and
5. the **umask** shell variable is reset to a default value (as distributed, 022).

Further, all subprograms are invoked as named in the configuration file, so if full path names are provided, the **PATH** variable will not be used. **This is highly recommended.**

If desired, the above behavior can be modified by use of several controls in the configuration file. The control

```
umask 077
```

will cause the **umask** to be reset to the named value, *which is given in octal*, rather than the default value. The control

```
environ SHELL=/bin/csh
```

says to pass the **SHELL** environment variable passed to the subprocess, and to give it the value "/bin/csh". If the value is omitted, the variable has the null value; and if the variable itself is simply named, the value from the environment in which *passwd+* is run is inherited. So, if a user uses the Korn shell, and **SHELL** is set to "/usr/bin/ksh", the line

```
environ SHELL
```

causes the subprocess to have the **SHELL** environment variable's value set to "/usr/bin/ksh".

As a final note, the help and password checking programs are invoked directly and not through an intermediate shell; so if a shell variable is to be used in the program name, the name must be passed to a shell. As an example, if every user had a program *checkme* in his or her home directory, to force that to be invoked, the line

```
environ HOME
```

would need to be present, and the command itself would be given as

```
/bin/sh -c “$HOME/checkme”
```

in the appropriate control line.

2.8. Miscellaneous Configuration Control Commands

The configuration file has several miscellaneous control statement. Because *passwd+* is designed to be portable, it uses library routines to access the password file. These routines, supplied with *passwd+* or written specifically for the system in use, use a common interface that is used to determine the account name, hashed password, UID and GID, login shell, home directory, and the relevant aging information. Currently, support for Berkeley 4.3 and Ultrix V4.2A password files are supported; to use the former, give the control

```
pwdfunc bsd4_3
```

and for the latter, replace `bsd4_3` with `ultrix`. Other types will be added soon, as need (and/or contributions) permit.

Three types of help are available; each default message may be replaced by the contents of a file. The first (nicknamed the “message of the day” file) is printed whenever *passwd+* is invoked, for all interfaces. The control line which sets this file name is:

```
            motdfile /etc/passwd.motd
```

The second file is printed whenever the option **-help** is given on a command line; it contains information about how to run *passwd+*. The file is printed, and the program then exits. The relevant control line is:

```
            interhelpfile etc/passwd.ihelp
```

The third file applies only to the interactive interface, and is printed when the user requests help (with the *help* command). The relevant control line is:

```
            helpfile /etc/passwd.help
```

In addition, a special help command can be run; typically, this is the password analysis program with a special option to print messages suggesting guidelines for choosing a good password. The line controlling this is:

```
            helpcheck /etc/pwcheck -help
```

If any of the help files are not present, a default help message is printed.

Finally, an extensive logging ability allows the system administrator to log messages about syntax errors in the configuration file, system errors (such as inaccessible files), information on the program’s use and on the success or failure of the use. This information may be stored in a log file, given as input to a command, written to a *syslog*(8) daemon, or printed on the standard error. For example, the line

```
            log system syntax “|mail staff”
```

sends all syntax and system error messages to all members of the mail alias *staff* (the last string means that the log messages generated from this line are input to the mail program), and the line

```
            log use result >/etc/passwd.log
```

writes messages indicating all invocations and their results into the log file “/etc/passwd.log”.

2.9. Example Configuration File

An example configuration file is given in Figure 1. Note that the pattern matcher used is the GNU emacs pattern matcher; given the patterns in the getgecos statement, this could equally well have been the Berkeley pattern matcher. In that same block, note that there is a getgecos line with a pattern that matches anything. Were this omitted, if none of the patterns matched the GECOS information field, the user would not be prompted for anything.

3. Password Checking

The heart of the password validation scheme is the program to verify that the password is in fact difficult to guess. The principles behind this program, which is invoked by *passwd+*, have been described elsewhere [1]. This section discusses *pwcheck*, the password checking program which is intended to be the password analysis component used by *passwd+*. Like *passwd+*, it uses its own configuration file.

3.1. Configuration File

As with *passwd+*, *pwcheck* uses a configuration file to enable system administrators to define the notion of an easy to guess password without writing or altering a program. When *pwcheck* begins, it reads its configuration file (or the standard input); when it reaches the end of the configuration file (or of standard input), it terminates.

The configuration file consists of control lines and test lines. The control lines set internal variables, interpolate files, and so forth; the test lines define tests and messages.

The basic unit of the little language is the *string*, which is defined as:

- a maximal sequence of alphanumeric characters and underscores “_”;
- a double quote ‘”’, left brace “[“, left curly brace “{“, dollar sign “$”, or at sign “@” followed by any number of characters (except a newline) terminated by the first unescaped double quote, right brace “]”,

```
# help files
motdfile            /etc/passwd.motd                  # new message of the day file
helpfile            /etcf/passwd.help                 # new help file
# who can do what and to whom
novalidate root :all:                                 # root can change anything without validation
validate :password: :self: :self:                     # validate user to change own password
novalidate :gecos: :shell: :self: :self:              # don't validate user to change shell or gecos fields
# shell control
getusershell                                          # allow any system shell
shell /usr/bin/ksh                                    # allow this (unlisted) one too
anyshell root                                         # what shell for root? anything she wants ...
ownshell                                              # users can make their own poison (anything they own)
# GECOS information -- two forms allowed:
pattern gnu                                           # GNU pattern matcher is really nice!
# form #1:          name,office,work phone,home phone,sponsor
getgecos "^\([^,]*\),\([^,]*\),\([^,]*\),\([^,]*\),\(.*\)$" name office home_phone work_phone sponsor
setgecos "%s,%s,%s,%s,%s" name office home_phone work_phone sponsor
# form #2:          name,contact (contact is who is responsible for that account)
getgecos "^\([^,]*\),\(.*\)$" name contact
# form #3           not allowed; this is anything else, and we want it put into form #2
getgecos "^\(.*\)$" name
setgecos "%s,%s" name contact
# now the prompts for all these billions and billions of variables (well, all 6, anyway ...)
associate name "Name" none                            # the name of the account holder
associate office "Office" "300 Bradley Hall"          # the office (default is department office)
associate work_phone "Office phone number" "(603) 646-2415" # office phone (default is dept number)
associate home_phone "Home phone number" unlisted # home phone (unlisted by default)
associate sponsor "Faculty sponsor" none              # who's sponsoring this account
associate contact "Contact person" none               # who is responsible
# subprograms; we're very restrictive here (guess why?)
umask 077                                             # give them NOTHING
environ SHELL=/bin/sh                                 # they got the Bourne shell, like it or not
environ PATH="/usr/bin:/bin:/usr/ucb"                 # a SAFE search path
check /etc/pwcheck                                    # the password checker ...
helpcheck /etc/pwcheck -help                          # and how to get help from it
onerror reject                                        # when in doubt, just say no!
# it's an ultrix system, so ...
pwfunc ultrix
# how and what do we log
log syntax system "|mail staff"                       # bad errors get mailed to anyone who can fix them
log use result syslog                                 # log use and success/failure to syslog daemon
```

Figure 1. A sample passwd+ configuration file.

right curly brace "}", dollar sign, or at sign, respectively; or

- any single character except alphanumerics, underscore, left and right braces, left and right curly braces, double quotes, dollar signs, and at signs.

Let us now look at how these strings may be used.

3.2. Setting and Referencing Variables

Unlike the previous version of *passwd+*, no variables are assigned values automatically. Instead, these values must be set by using explicit control lines. Also unlike the previous version, variable names may have

% [:] [_] [+-] [*m.n*] [^*|#] *variable*

:	insert \ characters as necessary to ensure the characters are interpreted as characters and not as metacharacters.
_	treat the value as an integer and subtract that value from the maximum password length. If the value is not an integer, 0 is interpolated.
+-	if "-", reverse the value; if "+" or omitted, do not reverse the value.
m.n	The characters beginning at position *m* (positions begin with 1) and through position *n* inclusive are interpolated. If "." is present, *m* and *n* are position 1 and the last position in the string, if omitted; if "." is not present, the first *m* characters are interpolated.
^*#|	^ makes all alphabetic characters in the value upper-case * makes all alphabetic charactes in the value lower-case | makes the first character in the value upper-case if it is alphabetic; no effect if not # interpolates the length of the string

Figure 2. The escape sequence components for accessing a variable value. All are applied right to left, and are applied *before* the result is interpolated.

---

more than one character in them.

The control

```
            set p bishop
```

sets the internal variable `p` to the (string) value "`bishop`". Similarly, variable values may be assigned using patterns; for example, the control line

```
   setpat "Matt Bishop* wrote 6 39" "^\(.*\) wrote \(.*\)$" who what x
```

would assign the value "`Matt Bishop*`" to the variable *who*, "`6`" to the variable *what*, and "`39`" to the variable *x*. As with getgecos, the interpretation of the pattern is controlled by

```
            pattern bsd
```

(for the Berkeley pattern matcher; to get the GNU *emacs* version, replace `bsd` with `gnu`).

Variable values are accessed and manipulated using an escape facility reminiscent of that of *printf*(3); see figure 2. For example, suppose *p*, *who*, *what*, and *x* have the values shown above. Examples of how the values may be accessed in different ways are:

```
%p        bishop        %3.5p      sho            %-2.4p    hsi
%(who)    Matt Bishop*  %:3.(who)  t Bishop\*     %^(who)   MATT BISHOP*
%_(who)   0             %_(what)   2              %-x       93
```

Note the value is *always* considered a string.

These particular operations were chosen because they are the most common ones password crackers use. The motive for the ":" was different, though. Variables are used in tests, and sometimes tests involve the evaluation of patterns. In these cases, some values are to be treated as strings (such as the password) and others as patterns (such as a set of variables containing patterns to be used repeatedly). So, when a variable value is to be treated as a string, it should have the ":".

One special variable has a value that changes. The sequence %< takes the value of the next line of the standard input, or the empty string if none. This is used to interact with programs which pass information to *pwcheck* using the standard input. For example, here is a sequence of controls that sets variables corresponding to the information passwd by *passwd+*:

```
            set newpasswd %<
            set oldpasswd %<
            set name %<
            set hashedpwd %<
            set Gecos %<
            set shell %<
            set home %<
            set UID %<
```

test	::=	'(' *test1* ')'	true if *test1* is true
	|	'!' *test1*	true if *test1* is false
	|	*test1* '&' *test2*	true if both *test1* and *test2* are true
	|	*test1* '|' *test2*	true if either *test1* or *test2* is true
	|	number1 '==' number2	true if number1 and number2 are numerically equal; any of the relations >, <, >=, <= != may be used here
	|	*string1* '==' *string2*	true if *string1* and *string2* compare equal; may use != here also
	|	*string1* '=~' *string2*	true if *string1* matches the pattern *string2*; may use !~ here also

In the above, *string1* and *string2* are both strings:

string	::=	string	a sequence of alphanumeric, underscore, and escaped characters
	|	'[' *filename* ']'	the string is any line in the file *filename*
	|	'@' *dbm_base_file* '@'	the string is in the *dbm*(3) file with base name *dbm_base_file*
	|	'$' *sorted_file_name* '$'	the string is any line in the sorted file *sorted_file_name*
	|	'{' *command* '}'	the string is any line in the output of *command*

Figure 3. Syntax for the boolean expressions in the test lines.

```
set GID %<
set nxtchange %<
set mustchange %<
```

Finally, to erase a variable value, use `unset`; the following erases the value of the variable *what*:

```
unset what
```

3.3. Tests

Tests are used to determine whether or not the password is suitable for use. Each test has four components: a boolean test expression (which evaluates to true or false), a true response, a false response, and a help response. For example, this sequence might be used to test password length:

```
length test %#p < 6
length true "%p" is unacceptable; you need at least 6 characters
length false the password passes the length test
length help Passwords must be at least 6 characters long.
```

Suppose the password is "`bis*hop`". In this case the boolean expression on the line beginning with "`length test`" is false, so the message on the line "`length false`" is printed. Note that the sense of the tests is that if the test succeeds (evaluates to true), the password is easy to guess. Also notice that variables may be included within the messages; they will be evaluated before the message is printed.

The word "`length`" at the beginning of the line serves simply to link the messages with the test. When a test is evaluated, its result is stored with the label at the beginning of the line (here, "`length`"). Whenever a line beginning with that label is encountered, it is processed. If the control ("`true`" or "`false`") matches the value of the test, the message is printed; otherwise it is ignored. If the test has not yet been seen, the messages are not printed. If labels are omitted, a singe (blank) label is assumed.

The line beginning with "`length help`" is a help line, and that is printed when *pwcheck* is invoked in help mode (by giving the command line option **-help**). In this mode, no test processing is done; variables are evaluated, and any test message lines with the control "help" are printed. This facility is to allow the system administrator to provide guidance on what passwords are acceptable.

In what follows, we refer to the boolean expression which is evaluated as the "test." The complete syntax of the tests is shown in figure 3; basically, all numerical tests and arithmetic operations are allowed, as are string compares and pattern matches. In addition, *pwcheck* can look for a password in a file or in the output of a program.

The tests are composed of numerical expressions, string compares, and pattern matches. For example, if `l` is the user's login name and `p` the proposed password, the test

```
"%p" =~ "\(%l\)*" | "%p" =~ "\(%-l\)*"
```

is true if the proposed password is 0 or more repetitions of the login name or the login name reversed (this requires the GNU *emacs* pattern matcher).

The use of files and programs is very similar. In the test

```
            [ /usr/dict/words ] == “%p”
```

each line of the system dictionary file “/usr/dict/words” is compared to the proposed password; if any match, the test succeeds.One problem with this test from the implementation point of view is that the search is linear and hence on a large file can be very slow. If the lines of the file are in sorted ASCII order, the above test may be rewritten as

```
            $ /usr/dict/words.sorted $ == “%p”
```

and *pwcheck* will use the binary search technique to locate the right side (the value of the variable *p*) in the file “/usr/dict/words.sorted”; note that if the file is *not* in sorted ASCII order, the result returned may be wrong. Finally, if the file is in the fast database format of the *dbm*(3) or *ndbm*(3) library routines, the command

```
            @ /usr/dict/words.dbm @ == “%p”
```

will use the *dbm* functions to search the file for the value of the right side (here, the value of the variable *p*). Note that the base name of the *dbm* files is given; in the above example, the files “/usr/dict/words.dbm.dir” and “/usr/dict/words.dbm.pag” must exist or the test fails.

The ability to use the output of a subprogram is a very powerful feature of *pwcheck*. For example, consider a test to catch all English words. If the proposed password is stored in the variable *p*, it would seem that

```
            [ /usr/dict/words ] == “%p”
```

would work, as “/usr/dict/words” is an on-line copy of an English dictionary. But note that some words in that file are capitalized, and others are not; in particular, “water” is in the dictionary, but if the proposed password is “Water”, the test fails. So perhaps

```
            { tr A-Z a-z < /usr/dict/words } == “%*p”
```

is better. This test runs the command

```
            tr A-Z a-z < /usr/dict/words
```

(which simply copies the contents of “/usr/dict/words” to the standard output, changing every capital letter to lower case). Again, this misses plurals, present and past participles, and other derivative forms of words. Only the spelling checker, *spell*(1), will catch these. So a command of the form

```
            { echo “%p” | spell } == “”
```

will work. (Recall *spell* prints on its standard output a list of incorrectly spelled words.)

Alas, while this does work, there is a serious security problem: the new proposed password will be passed to *spell* using the command *echo*(1), and for a brief time will therefore be visible to programs which can examine command line argument lists. To overcome this danger, *pwcheck* provides a special construct to send input to a program in a test. The above test should be written as:

```
            { spell } <- “%p” == “”
```

Here, the first string following the `<-` is written to the command’s standard input. This solves the above security problem, and will reject any English words.[1]

3.4. Miscellaneous Commands

The line

```
            include filename
```

interpolates the contents of “*filename*” at this point in the file. Files may be nested as deeply as the system allows (usually 16, 28, or 60). This is useful for including per-user tests; for example, if the user name is stored in the variable `l`, then

```
            include /etc/passwd.users/%l
```

1. Note that some versions of *spell* record misspelled words so a systems administrator can examine them and decide whether to add them to the dictionary or to a site-specific dictionary. If this is done, be sure to turn it off in the test! (This often can be done with a command option such as **-N**.)

```
# this is a sample password configuration file for pwcheck
set version beta-test
pattern gnu
# this file loads the variables (see section 3.2 of this paper)
include /etc/passwd.siv
#================= some stuff to make patterns easier to read
set let \[A-Za-z]
set nlet \[^A-Za-z]
#================= fix up first, last name, etc.
setpat “%(Gecos)” “^\([^,]*\),\([^,]*\),\(.*\)$” name office extension
setpat “%(name)” “^\(%(let)*\)%(nlet)*%(nlet)\(%(let)%(let)*\)%(nlet)%(nlet)*\(%(let)%(let)*\)$” \
                                        first middle last
setpat “%(name)” “^\([A-Za-z]+\)[^A-Za-z]+\([A-Za-z]+\)$” first last
set initials %1.1(first)%1.1(middle)%1.1(last)
#================= test password length
length test %#p<=6
length true “%p” is unacceptable; you need at least 6 characters
length false the password passes the length test
length help Passwords must be at least 6 characters long
#================= test if password is an English word
English test {spell} <- “%p\n” == ““
English true “%p” is unacceptable; it is an English word
English false the password passes the no-English-words test
English help Passwords must not be an English word
#================= test login name
login test “%p” =~ “\(%l\)*” | “%p” =~ “\(%-l\)*”
login true “%p” is unacceptable; it cannot be your login name repeated
login false the password passes the repeated login name test
login help Passwords must not be repetitions of your login name (or reversed login name)
#================= sorted dictionary (ASCII order)
dictionary test $/usr/dict/words.sorted$ == “%p”
dictionary true “%p” is in the system dictionary
dictionary false the password is not in the system dictionary
dictionary help Passwords must not reside in a system dictionary
#================= DBM file
dbmdictionary test @/usr/dict/words.dbm@ == “%p”
dbmdictionary true “%p” is in the system’s dbm dictionary
dbmdictionary false the password is not in the system’s dbm dictionary
dbmdictionary help Passwords must not reside in the system’s dbm dictionary
```

Figure 4. A sample pwcheck configuration file.

will include a file specifically for the current user.

Finally, although UNIX passwords may be of any length, only the first 8 characters are significant. Hence strings which differ in the ninth character are really the same so far as the UNIX password system is concerned. When the control

```
complen 8
```

is used, all string comparisons are done only for the first 8 character. (Any transformation to variable values are done first; the truncation occurs only during the comparison and only for purposes of the comparison.)

3.5. Example Configuration File

An example configuration file is given in Figure 4. Note that variables may contain patterns; however, in the line

```
set let \[A-Za-z]
```

the first backslash is needed. Were it not there, the "`[`" and "`]`" would mean the characters between were a file name, and the first line of the file would become the value of the variable `let`.

The assumption made about the GECOS field here is that it is of the form

name, *office*, *extension*

(just as the RIACS format shown in section 2.5). Note the use of the variables *let* and *nlet* in the patterns.

4. Future Directions and Work

The password changing program, *passwd+*, is stable; no major changes to the design have been made for some time, and it is in the process of being cleaned up and documented so that it can be distributed for beta test. The thrust of future development will most likely be in developing library routines to handle different password storage schemes and password distribution mechanisms.

The password checker is another matter. It is undergoing changes in both design and syntax. The program *pwcheck* was designed to be independent of *passwd+*, and as such lacks some features and transformations which would be of great use, such as the ability to count letters, non-letters, and so forth. These may be done using other programs such as *sed*(1) or *awk*(1); however, this requires a subprogram be run, and is very awkward. Hence it is clear these abilities need to be added, but it is not clear how they should be integrated with the rest of *pwcheck*.

Part of the problem is that little, if any, research has been done on language design for this purpose; this suggests that perhaps too little research on the usefulness of proactive password checking itself is documented. Logic and intuition says that proactive password checking is far more effective than password cracking, and should be as good as (if not better than) other forms of password assignment. No experimental evidence has been gathered to support this opinion, and human nature being the crux of the problem, it would be very wise to test these beliefs.

5. Conclusion

The goals of *passwd+* and *pwcheck* are to provide a friendly, powerful tool for system administrators to improve the quality of the passwords users select. Sufficient facilities are provided to allow per-site and per-user tests.

The software should be available soon; its location will be announced through the *cert-tools* mailing list when it is available. The version which will be released will be beta test software, so people who take it will be asked to report bugs (and fixes), as well as any new password file routines they write. Documentation on the software and its internal structure will be available to help any hardy souls willing to work with it!

6. References

[1] Matt Bishop, "A Proactive Password Checker," in *Information Security*, David T. Lindsay and Wyn L. Price (eds.), North-Holland, New York, NY pp. 169-180 (1991)

[2] M. Gasser, "A Random Word Generator for Pronounceable Passwords," Technical Report ESD-TR-75-97, The MITRE Corporation, Bedford, MA (Nov. 1975)

[3] L. Grant, "DES Key Crunching for Safer Cipher Keys," *SIG Security Audit and Control Review* **5**(3) pp. 9-16 (Summer 1987).

[4] J. Haskett, "Pass-Algorithms: A User Validation Scheme Based on Knowledge of Secret Algorithms," *Communications of the ACM* **27**(8) pp. 777-784 (Aug. 1984).

[5] Daniel V. Klein, ""Foiling the Cracker": A Survey of, and Improvements to, Password Security," *Proceedings of the UNIX Security Workshop II* pp. 5-14 (Aug. 1990)

[6] Robert Morris and Ken Thompson, "Password Security: A Case History," *Communications of the ACM* **22**(11) pp. 594-597 (Nov. 1979)

[7] UNIX User's Reference Manual, 4.3 Berkeley Software Distribution Virtual VAX-11 Version, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering

and Computer Science, University of California, Berkeley CA (Apr. 1986); reprinted by the USENIX Association (June 1987).