A PROACTIVE PASSWORD CHECKER

Matt Bishop

Technical Report PCS-TR90-152

June 1990

A Proactive Password Checker

Matt Bishop

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

ABSTRACT

Password selection has long been a difficult issue; traditionally, passwords are either assigned by the computer or chosen by the user. When the computer does the assignment, the passwords are often hard to remember; when the user makes the selection, the passwords are often easy to guess. This paper describes a technique, and a mechanism, to allow users to select passwords which to them are easy to remember but to others would be very difficult to guess. The technique is site, user, and group configurable, and allows rapid changing of constraints imposed upon the passwords. Although experience with this technique has been limited, it appears to have much promise.

1. Introduction

Computer generation of passwords is a delicate art: the passwords cannot be random, for if they are users will write them down, yet they cannot be too non-random, for otherwise they could be easily guessed. The use of pronounceable passwords tries to strike the balance, yet even these are often difficult to remember. Hence user selection of passwords is in widespread use; unfortunately, many users select easy to guess passwords. In a series of experiments in 1979, roughly 33% of the passwords appeared in a dictionary and an astonishing 86% of all passwords were guessed [11]; in similar experiments conducted last year at the Software Engineering Institute, over 21% of the passwords from many different sites were guessed correctly within one week, and roughly 3% were found in 15 minutes [9]. Clearly, this is as unacceptable as users writing down their passwords.

Two styles of passwords are in use today. The older is the traditional assignment of a password to an account or role; to access that account or role, the user supplies the password for authentication. The newer is the use of "variable" passwords, including both the one-time password which changes with every use (for example, a smart card), and the challenge-response protocol, in which the user is *challenged* by the system to provide proof of identity, and the user's *response* is based upon the specific challenge issued by the system.

In this note we assume the site cannot use a variable password scheme, and consider an improvement to the traditional scheme. We introduce a tool, called a *proactive password checker*, designed to ameliorate this problem at the source. When a user uses this tool to change (or set) a password, the proactive password checker runs a number of tests on the proposed password to see if it can be guessed easily; if so, the user is informed and the password is rejected. Thus, the tool allows users to select passwords they can remember, but prevents them from choosing passwords which are deemed unsafe.

The next section of this paper discusses current methods of assigning passwords, and describes some limits and constraints on them. The section after that enumerates various criteria that a good password selection technique must meet; we then describe an implementation of a tool that provides a balance between constraint and memorability. We summarize with a discussion showing the tool does meet the stated criteria, some experiences, and suggestions for improvements and future directions of research and development.

2. Previous Work

While such an idea is not new (indeed, most password changing programs perform some simple tests on length and character mix), the scope of the proposed tool is radically different in that it allows for an *arbitrary* set of tests, thereby giving each site administration the power to decide what passwords are "guessable" without having to reprogram or rewrite the password changer. By contrast, other password changers have their tests preprogrammed, so system administrators must accept the ones deemed suitable at the programming site rather than at their own site.

Perhaps the most obvious way to obtain "good" passwords is to have the computer generate them randomly. If this is done without constraint, one must consider the ability of human beings to remember those passwords. Miller's summary of experiments [10] indicate that about 8 "chunks," or meaningful items (such as digits, letters, or words) can be repeated with perfect accuracy; this means that at most one random password of 8 "chunks" (or two of 4 "chunks," and so forth) can be remembered correctly. Given that most people have more than one account, and so must remember more than one password, most users will write such passwords down, or use some memory aid that may very well be found and used by a third party.

An interesting suggestion to overcome the tendency of users to write down random passwords has been in use at one large installation in which many people must know the operator passwords to 40 (or so) computers. It essentially provides each user with a list of passwords altered using an invertible password algorithm. For example, if the algorithm were "capitalize the third letter and append the character x," and the operator's password for computer *athene* were listed as "gleork" then the real password would be "glEorkx" [6]. This technique is akin to more general "pass-algorithm" techniques mentioned earlier, but does not involve challenge-response protocols.

On the basis of Miller's summary, it has been suggested ([5], p. 342) that if the password were somehow made "meaningful," that is, if some association could be found to aid the user's memory, then the user could memorize considerably more, because the password would itself become a "chunk." One common technique is the use of a pronounceable password, which contains two or three chunks only; indeed, such passwords are recommended in many guidelines [7]. Perhaps the most sophisticated such generator was designed for Multics [8], which used a considerable statistical analysis of English to generate passwords. The problem with such schemes is that the phonemes – the smallest unit of pronunciation, and hence the "chunk" – are themselves essentially random, and therefore while this scheme is suited for one or two passwords, users will still write the passwords down when faced with remembering many different pronounceable passwords for many machines.

One general problem with computer-generated passwords is the use of a pseudorandom number generator to produce the passwords. If the period of the pseudo-random number generator is too short, the password can be easily compromised. Morris and Thompson give a very dramatic instance of this; an unnamed site generated passwords composed of eight random letters or digits. To test all 2.8×10^{12} possible combinations of those characters on the computer used would take 112 years. Unfortunately, the pseudo-random number generator used had a period of 2^{15}, so the 3.3×10^{4} possible passwords could be tested in about 41 minutes – which the attacker did [11].

The alternative to computer generation of passwords is to have the user select his or her password. As mentioned in the introduction, if the user is allowed to choose a password without restrictions, the choice may very well be poor. (We should note that the sites involved in the experiments placed simple restrictions of length and character mix on the passwords.) Hence, some restrictions are necessary; in the next section, we consider what the nature of the restrictions should be. First, though, let us examine what makes a password a "poor" choice.

3. Requirements

Exhaustive search for a password will always recover the password; our goal is to make such recovery as expensive as possible. This implies that an "unguessable" password is one for which any systematized search for the password is no better than a random search for the password.

Note that this does *not* mean that the password the user chooses must be as unlikely as possible, which would imply some special ordering of passwords in order of probability; rather, the goal is to *eliminate* any such ordering, because should the attacker discover any such ordering, he could use that information to guide his search. So, ideally, no one potential password is a better choice than another [3].

Were it not for the fallibility of human memory, this would mean that computer generation of random passwords is best. Unfortunately, the limits on the number of "chunks" that a person can remember, combined with most people needing more than one password, preclude such generation. However, what may be a meaningful "chunk" to one user may be completely meaningless, and therefore *appear* random, to another; so, ideally, passwords should be composed of this kind of "chunk." Hence for our purposes, we can define the concept of "guessability" as being a measure of the apparent randomness of the "chunks" to parties other than the person with whom the password is associated.

Hence some obvious passwords can be ruled "guessable" very quickly. Names, dictionary words, cartoon characters, and so forth can be quickly guessed. Extending this slightly, simple transformations of such words should also be deemed "guessable;" for example, most on-line dictionaries do not contain plurals formed by adding "s" to the singular. Yet these should be rejected, as a common method of generating additional guesses is to add an "s" to nouns in a dictionary.

More subtle are the passwords that seem random to all but a subset of users, or that are meaningless unless external data is known. For this reason, the determination of whether or not a password is guessable must use site-specific (or group-specific) and user-specific knowledge. For example, the password "mhmhdcms" is, on first glance, a random collection of lower-case letters. However, if the user worked at Mary Hitchcock Memorial Hospital, the Dartmouth College Medical School's teaching hospital, the password becomes fairly obvious and would be one an attacker would try. As another example, the author's password "Heidiu´"[1] would be difficult for a random attacker to guess, unless he knew the author's daughter were named "Heidi Tinúviel;" then, the ú is distinctive enough so a reasonable guess would be to append "u´" to her first name.

These constraints suggest that current password changing programs are not rigorous enough, because they apply fixed tests based either on length, character mix, limited personal data, or word lists. If a word meets the length and character mix requirements, and is not produced by

1. Obviously, this is not his real password.

the specific set of programmed transformations on the personal data or word lists, it is accepted and made the new password. The problem is that first, these password programs suffer from programming limitations (for example, they use only one word list, or do not allow the word list to be chosen on a per-user basis) and second, changing the transformations is quite time-consuming, and may be put off if other matters are deemed more important (for example, programming a test to check for the present participle of verbs is actually quite tricky, yet this is a fairly obvious transformation for an attacker to try).

Bearing this in mind, the only way to enforce these criteria in a usable, administrator-friendly fashion is to have a configuration file containing tests that the password checker will use to determine the guessability of any proposed password before it is accepted; should the proposed password be deemed "guessable," it will be rejected. The use of a configuration file allows the rapid changing, or addition, of tests without having to recompile or relink the program; also, by designing the language encoding the tests properly, it would be able to use subprograms (such as the system spelling checker) to test a proposed password against many more English words than are in the system dictionaries. Finally, as the configuration file can be altered on a per-site basis, and tests can be based on user identity, it provides the ability to tailor the tests to both the system environment and the user.

The use of a configuration file also suggests that error messages can be associated with each test, so that if the proposed password passes the test (and hence is "guessable"), an *informative* error message, stating the precise reason for rejection, is printed. Such messages would quickly educate users on how to choose a good password, and would produce less hostility than messages of the form "password invalid – no change."

We now describe the design and implementation of a UNIX[2]-based version of this tool, which meets all of the requirements above.

4. Implementation of a UNIX Version

The algorithm used by the proactive password changer is straightforward: the user is asked to type his current password, which is then validated, and is then asked to type the proposed new password twice (to eliminate typing errors). If the two are the same, the configuration file is read and its commands executed; these include commands to obtain information about the user, about the system, and tests through which the proposed password is run. If the proposed password passes

2. UNIX is a Registered Trademark of AT&T Bell Laboratories.

any of the tests, it is deemed too easy to guess and is rejected. (In this case either the error message associated with the test, or a standard error message, is printed.) If the proposed password fails all the tests, it is accepted and the user's password changed.

The novel feature of this proactive password changer is the use of a very flexible, very powerful interpreted little language for representing the tests. The little language describes tests with associated error messages; each test is composed of any combination of variables, constants (numbers and strings), equality and pattern matching relations, and file and program output scanning instructions.

Variables are set in one of three ways: either by the program at start-up, by the system administrator, or by analysis of a system database; the latter two are explicitly done in the configuration file. Variables are of type *number* or of type *string*, and their value is referenced by prefixing the variable's name with "%". For example, the variable "u" contains the user's name, so

```
%u
```

accesses the user's name. Note that the access takes the form of textual substitution.

At times, only part of the value of the variable may be useful; for example, one could test for the top-level domain of the host. In this case, only the last 3 characters of the value of the variable "d" are desired. To make obtaining them simple, several optional feilds may be added to the variable reference. These take the form

% [–] [*b*] [. *e*] [*f*] *v*

where the quantities in brackets "[ ]" are optional, and mean:

– If the variable is a string variable, then its value will be reversed; if the variable is a number, its value will be the variable's value subtracted from the password's length

b This is ignored for number variables. For string variables, it denotes the first position of the returned value (the first character is at position 0); if omitted, it is the beginning of the value.

e This is ignored for number variables. For string variables, it denotes the last position of the returned value; if omitted, it is the end of the value.

f This is ignored for number variables. It describes how the value is to be transformed for string values. Legal symbols are: "^" to capitalize all letters, "*" to make all letters lower case, "|" to capitalize the first character if it is a letter, and "#" to be the length of the variable's value.

As an example, suppose we wished to determine the second through seventh characters of

name	*type*	*meaning*	*name*	*type*	*meaning*
0...9	string	defined in config file	n	string	user's full name
a	number	number of alphanumeric chars	o	string	user's office
b	number	number of alphabetic chars	p	string	proposed password
c	number	number of upper case chars	q	string	current password
d	string	domain name of host	s	string	user's surname
f	string	user's first name	t	string	user's phone number
h	string	host name	u	string	user's login name
i	string	user's initials	v	number	1 for mixed case, else 0
l	number	number of lower case chars	w	number	number of decimal digits
m	string	user's middle name(s)			

Figure 1. The variables are strings of characters or numbers and are accessed as "*%rb.efv*" where:
r reverse (string variable) or subtract the value from the length of the password (number variable)
b number of position of first character to access (string variable); ignored (number variable)
e number of position of last character to access (string variable); ignored (number variable)
f format: ^ capitalize all letters, * lower-case all letters, | capitalize first character if alphabetic, # value is length of string (string variable); ignored (number variable)
v variable name

---

the user's login name reversed. (This might be combined with a trailing digit to make a password which, for some reason, the site administrator does not wish to allow.) Then the reference to the variable would be "%-1.6u". Similarly, if the proposed password were "g&Un3", then "%p" is "g&Un3", "%#p" is 5 (the length of the password), "%a" is 4 (as the value of "p" has 4 alphanumerics), "%b" is 3, "%c" is 1, "%l" is 2, "%v" is 1 (as the value of "p" has both upper and lower case), "%w" is 1, and "%-p" is "3nU&g".

The assignment of user names exemplifies the ways variables can be assigned to. The simplest method is to use a "set" statement, as in

```
set: f "Matthew"
```

which sets the value of the variable *f* to the string "Matthew". More effective in this context is to use the "gecos" statement to parse the user information in the password database file.So, if that file contained the field "Matt Bishop,N230-102,6921" then the following line would assign "Matt" to *f*, "Bishop" to *s*, "N230-102" to *o*, and "6921" to *t*:

```
gecos: "%s %s,%s,%s" f s o t
```

If the format does not match the database entry, the line is skipped and successive gecos lines are tried until a match is found. As these database fields are very often inconsistent, this allows the system administrator to try a number of different formats.

Figure 1 summarizes the variables, their types, and their settings. Note that values of "a",

```
test      ::=  '(' test ')'            pass if test is passed
            |  '!' test                negate result of test
            |  test '&' test           pass if both tests pass
            |  test '|' test           pass if either test passes
            |  number '==' number      pass if the numbers are (arithmetically) equal
            |  exstring '==' string    pass if (any line of) exstring is identical to string
            |  exstring '=~' pattern   pass if (any line of) exstring matches pattern
            ;
exstring  ::=  string                  exstring is the given string
            |  '[' filename ']'        exstring is the set of lines in the file
            |  '{' program '}'         exstring is the set of lines of output
            ;
```

Figure 2. The BNF defining the tests. Note that when *exstring* is the contents of a file or the output of a program, the tests using it are iterated once for each line of the file. Also note that pattern matching is done using the pattern matcher in the system library; this is invariably the same one as used by the system editor *ed*(1) [1][4].

"b", "c", "l", "v", and "w" all depend upon the value of "p".

The syntax of the tests is summarized in Figure 2, and is very straightforward. Because of the complexity of most pattern matching expressions, the pattern matchers use the system's standard library routines to determine if a value matches a pattern; this means that a system administrator who uses a text editor need not learn a new pattern matching system. The file and program scanning instructions compare a value to the contents of a file or the output of any program, or match the contents of the file or output of the program to a pattern written in the style of the system's text editor. Comparison is done line by line, and if any line satisfies the relationship, the test succeeds. Variables in the file names and program commands are replaced before the files or commands are executed, so it is easy to condition the test, the files, and the programs upon the user's or group's identity.

Some example tests will show how constraints on acceptable passwords are implemented.

```
"%p"=~"[0-9][A-Za-z]{3}[0-9]{3}"
```

succeeds if the password looks like a California automobile license plate number (a digit followed by three letters followed by three digits); it matches the proposed password (`%p`) against an appropriate pattern;

```
"%*p"=~"\(%u\)*"
```

succeeds if the password is 0 or more repetitions of the user's login name; it first maps all alphabetic characters in the proposed password to lower case (`%*p`), then matches them against the pattern composed of the login name (`%u`) repeated 0 or more times;

```
"%^p"=="%-^l"
```

succeeds if the password is the same as the user's last name reversed; this test makes all alphabetic characters in the proposed password upper case (`%^p`), does the same for the reversal of the user's last name (`%-^l`), and tests for equality.

```
"%p"=~"^[a-z]*$"|"%p"=~"^[A-Z]*$"
```

succeeds if the proposed password is a monocase alphabetic sequence; and

```
(%#p==%b)&(%v==0)
```

also succeeds if the proposed password is a monocase alphabetic sequence, but by comparing the length of the proposed password (`%#p`) to the number of alphabetic characters in it (`%b`), and if the password is composed *only* of alphabetic characters, tests if they are all of the same case (`%v` is 1 if so);

```
[ /usr/words/dict ]=="%*p"
```

succeeds if the word is in the system dictionary (the file */usr/dict/words*). This does not, however, catch words like "`waters`" which are not in that dictionary.

```
{echo %p | spell}==""
```

succeeds if the word is an English word; the password is given as input to the system spelling checker; if found, the checker returns nothing. This would catch plurals, participles, and other transformations of the words in the system dictionary.

```
[ /etc/pwbad/%u ]=="%*p"
```

compares the password against the words in the user-specific word list named by the user's login and in the directory */etc/pwbad*. This is an example of how to exclude user-dependent information.

Error messages may follow each test; if any test succeeds, the appropriate error message is printed. This allows users to be told precisely why their selection is unacceptable, rather than limiting the response to a more generic "`password not suitable; try again.`" So the lines containing the above tests would be better written in the configuration file as:

```
"%p"=~"[0-9][A-Za-z]{3}[0-9]{3}"cannot use license plate numbers
"%*p"=~"\(%u\)*"                  cannot use (repeated) login name
"%^p"=="%-^l"                     cannot use last name reversed
```

```
"%p"=~"^[a-z]*$"|"%p"=~"^[A-Z]*$"cannot use small letters only
(%#p==%b)&(%v==0)                cannot use small letters only
[ /usr/words/dict ]=="%*p"       cannot use dictionary word
{echo %p | spell}==""            cannot use English word
[ /etc/pwbad/%u ]=="%*p"         cannot use user-specific information
```

and if, for example, a user tried to change his password to "1PLK109" the error message "`cannot use license plate numbers`" would be printed.

We should note that the "set" statement mentioned earlier may also use the output of a program or a file; in such cases, the variable is assigned the contents of the first line as its value. For example,

```
set: 0 { groups | tr ' ' ',' }
```

sets the variable "0" to a comma-separated list of groups to which the user belongs.

The configuration file provides one additional control relevant to password testing. On UNIX-based systems, only the first 8 characters of the typed password are significant [2]; hence, the typed password "ambiguous" is passed to the program as "ambiguou". If the naive approach to string comparison were taken, this word would fail the test

```
[ /usr/words/dict ]=="%*p"
```

even though the word "ambiguous" is in the dictionary and, in fact, is therefore easily guessed. To handle this, the password changer treats all string comparisons as testing for matches in the first 8 characters. The "sigchar" statement can be used to reset this number; for example,

```
sigchar: 6
```

causes the program to treat all string comparisons as testing for matches in the first 6 characters. If the number of significant characters is set to 0, the length of the password is used. (Note that on UNIX systems, the program will *never* obtain more than 8 characters from the user's typed password due to the way the library input routine works. Hence it makes no sense to increase this beyond 8.)

For compatibility with the Berkeley password changing program, the proactive password changer also has the ability to change the gecos (user information) field of the password database file. The "setgecos" statement works like the "gecos" statement, except that rather than assign values to the variables after the format statement, the user is prompted for the value; the result is then formatted using the format string and written to the gecos field. A “forcegecos” statement is provided for those cases where no “setgecos” statement format matches the format of the data in the gecos field; this statement prompts the user for the requisite information, which is then written to the file using the specified format.

A sophisticated logging mechanism is available to aid system monitoring, debugging, and test selection. In addition to any system messages (such as unavailable files or corrupt password files), the configuration file may be set to log:

- the user executing the program, the user whose password is being changed (note these two may be different if the superuser executes the program), and the password file containing the password being changed. This allows monitoring the frequency of changes, and can be used to implement a password aging scheme if desired.
- the success or failure of the attempted change and, if the latter, the specific test passed. This enables system managers to determine which tests are most useful and, if necessary, circulate a memo reminding users that passwords of certain types are easily guessed. Note that this; records neither the user whose password is being changed nor the rejected password.
- debugging information, indicating the success or failure of each test, the values of variables, and other useful information useful for debugging both the program and the tests. Given the amount of information produced, it is strongly recommended this mode not be used in production.
- syntax errors in the tests.

Logging information can be sent to any combination of the standard error output, a file, the input of a program, or the system logging facility. This flexibility allows syntax errors to be mailed to all system administrators, and user and success/failure information to be logged to the system logging facility. Hence system administrators can be notified of emergencies without being bothered by day-to-day information.

5. Experiences

Experience with this program is somewhat limited, but is very encouraging. It has been in use at the Numerical Aerodynamic Simulation facility’s workstation subsystem (composed of workstations including Suns[3] and IRISes[4]). No users have complained or commented about the

3. Sun is a trademark of Sun Microsystems, Inc.
4. IRIS is a trademark of Silicon Graphics, Inc.

password program, which indicates that either they have not changed their password for the past year or have not noticed the change (which was not publicly announced). This site periodically runs a password cracker which attempts to guess passwords using several very large dictionaries and word lists; to date, none of the users who have changed their password using the proactive password changer have had their passwords guessed correctly.

The most serious problem with the proactive password changer proved to be the dependency of the pattern matching scheme upon the operating system version. As it uses the underlying pattern matcher of the system, it is sensitive to differences in the UNIX System V pattern matching language and the 4.3 Berkeley Software Distribution UNIX pattern matching language; specifically, the configuration files for System V based systems are (usually) incompatible with the configuration files for 4.3 BSD based systems. Hence system administrators must be careful that their patterns in the tests are correctly written.

6. Conclusions

The experience so far appears to justify the claim that the concept of checking a proposed password for suitability before allowing it to become the password is at least as good as attempting to guess passwords after they have been changed. Thus far, when a reasonable configuration file is used, the passwords selected it have been, for all practical purposes, unguessable. This eliminates the necessity of badgering the user to change an easily-guessed password, or the resentment of that user when a system administrator changes the password for him. Further, there is a window of vulnerability between the first change (to an easily-guessed password) and the second (when the user or system administrator changes it) that an attacker could exploit; the proactive password changer eliminates this completely. But given human nature, logic is often eminently reasonable, completely unassailable, and dead wrong; so more testing is necessary to validate the claim that the proactive changer is as good as attempting to crack passwords after the change.

The concept being laudable, what can be said about the program implementing it? This first effort has several shortcomings, the major one being the incompatibility of configuration files across different versions of the operating system. Ideally, one should be able to specify in the configuration file which pattern matcher (System V or 4.3 BSD) is to be used; however, as both implementations are legally protected, this will have to await creation of public-domain versions. An alternative would be to use the GNU pattern matcher, but that would require system administrators to learn a new pattern matching language (except for those who use GNU software already).

The little language used to represent the tests is very primitive and should be expanded considerably. Specifically, inequality constructs should be added, and some other variables would make writing tests easier (for example, variables for the number of control and punctuation characters would be very useful). Also, currently each test can take at most one line; while the maximum line length is generous (currently 1024 characters), lines that long are unreadable and uneditable by most UNIX text editors, so a facility to split one test over several lines should be added. Finally, a capability to give input directly to programs is necessary. In the test

```
{echo %p | spell}==""
```

the proposed password is used as a command-line argument, making it potentially visible to the process status listing program *ps*(1). A construct must be added to eliminate this visibility.

The "gecos" and "setgecos" statements are not very sophisticated in their analysis of the user information; specifically, the trailing "Jr." or "Sr." in a name like "Bruce Partington, Jr." will cause "Bruce" to be treated as the first name, "Partington" as the middle name, and "Jr." as the last name. A more complex analysis routine should be added, or a better mechanism for analyzing the user information should be written.

Finally, the error messages currently do not allow any variables to be interpolated. The ability to do so would allow a more personalized, detailed error message which could be tailored to the particular rejected password as well as the test passed. This ability will be added in future implementations.

Acknowledgements: The research leading up to this work, and the work itself, were supported in part by grant NAG 2-628 from the National Aeronautics and Space Administration to Dartmouth College, and in part by a Burke Award from Dartmouth College. Thanks to Dan Klein of the Software Engineering Institute and Michele Crabb of Computer Sciences Corporation (contractor for the NAS Project), both of whom gave me excellent feedback on the implementation; to Michele and Robert Van Cleef of NASA, for installing and maintaining it on the NAS workstations, and to Dan, for encouraging me to write this paper, and having infinite patience waiting for it!

References

[1] *System V Interface Definition Issue 2, Volume 2*, AT&T Bell Laboratories, Indianapolis, IN (1986) pp. 55-64.

[2] M. Bishop, "An Application of a Fast Data Encryption Standard Implementation," *Computing Systems* **1**(3) (Summer 1988) pp. 221-254.

[3] M. Bishop, "Password Checking Techniques," *Proceedings of the Workshop on Computer Security Incident Handling* (June 1990) pp. III-D-1.

[4] *UNIX User's Manual Reference Guide, 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).

[5] C. Coombs, R. Dawes, and A. Tversky, *Mathematical Psychology: An Elementary Introduction*, Mathesis Press, Ann Arbor, MI (1981).

[6] M. Crabb, "Password Security in a Large Distributed Environment," *Proceedings of the UNIX Security Workshop* (1990), to appear.

[7] *Password Management Guideline*, Report CSC-STD-002-85, Department of Defense Computer Security Center, Fort George G. Meade, MD (Apr. 1985).

[8] M. Gasser, "A Random Word Generator for Pronounceable Passwords," Technical Report ESD-TR-75-97, Electronic Systems Divisaion, Hanscom Air Force Base, Bedford, MA (Nov. 1975).

[9] D. Klein, "Foiling the Cracker: A Survey of,and Improvements to, Password Security," *Proceedings of the UNIX Security Workshop* (1990), to appear.

[10] G. Miller, "The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review* **63** (1956) pp. 81-97.

[11] R. Morris and K. Thompson, "Password Security: A Case History," *CACM* **22**(11) (Nov. 1979) pp. 594-597.