Proactive Password Checking

Matt Bishop

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

ABSTRACT

This paper presents generic requirements for proactive password changer. It then examines two of the most popular, publicly-available programs to see how well they meet the requirements. Future directions are examined, as well as alternatives to such checkers.

1. Introduction

The problems inherent in allowing users to choose passwords without restriction have been widely discussed [4][6]; countermeasures include random generation of passwords [8] and techniques to test the strength of the proposed password the user selects. The latter falls into two classes: reactive password checking, in which the password is reset and later tested, and proactive password checking, in which the proposed password is first tested and then made the password. The manner of testing is independent of class. Either the password is transformed into the form in which it will be stored and a program that attempts to guess the password (a password cracker) is invoked, or the password is analyzed before it is transformed.

On computers running some version of the UNIX™ operating system, reactive password checking requires the former method, as the password is stored as a non-invertible hash; hence analyzing the structure of the password requires determining what that password is, and if this can be guessed by a program, an attacker can probably guess it too. Proactive password checking allows a combination of either testing method, although the pre-transformation analysis is probably more prevalent.

This paper discusses the requirements for an effective proactive password *checker*. We limit ourselves to the testing of the password, because those requirements will be independent of the system on which the checker runs. That portion of the program which does the actual altering of the password, which we shall call the *changer*, depends heavily on system-specific information and custom (for example, allowing the superuser to change anyone's password without validation is customary in non-secure versions of the UNIX operating system, but one can easily design a password changer which does not allow this). For the purposes of this discussion, we shall assume that the user's password is somehow obtained and given to the password checker, along with other relevant information; our concern is what should be looked for to determine how easily the password may be guessed.

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

2. Analysis and Requirements

Let us first outline the problem of easily guessed passwords. Let *P* be a set of words from which a password *p* is to be selected. Let *p'* be a guess for this password. Assume it takes a constant amount of time $T = t(p')$ to determine if this is correct. Let *G* be the random variable representing the password, and let *H* be the random variable corresponding to the time to guess the password. Then, assuming the distribution function of *G* is uniform, the expected time to guess a password is:

$$A \equiv E(H) = \sum_{i=1}^{|P|/2} T = T\frac{|P|}{2}$$

assuming no prior knowledge of the function used to do the authentication[1].

For any distribution function of *G*, we shall call a password *easily guessable* when $E(H) \le kA$; that is, when the distribution function of *G* makes it significantly more likely that a password will be guessed, than if the distribution function of *G* were uniform. (In other words, the distribution of passwords is skewed to such a degree that it is easier to guess a password than if there were no skew.) The parameter *k* controls the degree of skewing that is considered acceptable, and depends as much on the site requirements as the set of possible passwords. If $k \ge 1$, for example, the set of legal passwords is unsatisfactory because they all can be easily guessed. In what follows, we assume that *A* is very large.

Let *s* be the function used to select a password from the set *P*; for example, *s* could be equally likely to select any element of *P* (in which case the computer would be generating random passwords), or *s* could allow the user to control selection (which is what we are considering). This selection function induces a probability distribution upon elements of *P*. With the first (random) selection function, this distribution is uniform; with the second, tests have indicated that the distribution is highly skewed. Hence by appropriately ordering the elements of *P*, one can substantially decrease the expected time to guess a password – often to the point where a password chosen from a subset of *P* could be guessed in less than time kA.

Two approaches can reduce or eliminate the skew. The first is to change the selection function to force a probability distribution that has some known, desired distribution; generation of passwords using a uniform random number generator does this. The second is to artificially delete from *P* those elements creating the skew; in other words, disallow passwords which have a high probability of being selected. This induces a more uniform, and hence more desirable, distribution. This is what proactive password checking is designed to do.

We should note that this analysis highlights a fundamental problems of proactive password checking. Were the checker to be overly restrictive in its definition of "high probability", it could eliminate so many potential passwords that any of the remainder could be

1. For example, if it is known that one potential password is correct if and only if another is correct, there is no need to test the first once the second is tested; this would alter the above equation somewhat.

guessed quickly. Hence some care must be applied in selecting which passwords are unacceptably easy to guess.

Turning from the mathematics to the engineering aspect, certain facilities must be present to provide the degree of flexibility in the tests that will eliminate passwords as easily guessed. Previous studies, notably [4] and [6], have described common classes of passwords found experimentally and what types of passwords should be avoided (see table 1);

Table 1: Passwords To Be Avoided

transformations of user's account name	transformations based on user's real name
words in a dictionary (English and foreign words, male and female names, etc.)	words which match a dictionary word if all capital letters are made lower-case
a word with a character turned into a digit or control character (*eg*., 's' to '5' or 'a' to '^A')	conjugations, participles, plurals, and/or possessives of words in dictionaries
appending or prepending numbers	keyboard patterns (*eg*., 'qwerty' or 'zxcvb')
reversed words in a dictionary	reversed words which match a dictionary word if all capital letters are made lower-case
short passwords	alphabetic-only or monocase passwords
numeric passwords (like Social Security numbers or telephone numbers)	license plates (not just of one state)

in addition, specific words or character sequences may be meaningful to particular sites or users and hence good guesses for attackers. These considerations suggest the following requirements for a proactive test mechanism:

1. the test mechanisms should be invoked on every proposed password, and no proposed password failing tests should be allowed to become a password;
2. the test mechanism must be able to reject any password in a class of common passwords (the classes being known from the literature or from experience)
3. the tests must allow per-user discrimination (that is, a password may be unacceptable for one user but acceptable for another);
4. the tests must allow per-site discrimination; and
5. the tests must allow some form of aging, so that passwords cannot be re-used within "too short" a period of time ("too short" being determined by the system administrator).

From these, we can glean additional recommendations for proactive password checking programs:

6. setting up the tests must be straightforward, because complexity breeds errors.
7. do not require all potential passwords to be in dictionaries; creating a comprehensive dictionary may take a considerable amount of time and space, and will still not include enough passwords. This suggests some form of pattern matching would be very appropriate.

8. do allow the proposed password to be run through a program, and act based on the output (or lack of output) of that program. This allows a password cracker to be used if desired; it also allows programs such as the spelling checker to determine if a proposed password is a word.

Finally, the issue of user friendliness must be addressed. By the principle of psychological acceptability [11], the proactive password checker must not make changing the password more difficult than changing the password without using the checker. This means that if the checker rejects a password, it should explain exactly why the proposed password is unsuitable, so that the user can learn how to pick an acceptable password. If this happens with too many potential passwords, the system administrator should check that the tests are not overly restrictive, and eliminating too many potential passwords.

3. A Comparison of Some Proactive Password Checkers

In this section, we compare several proactive password checkers to the requirements and recommendations above. We confine our comparison solely to the password checking part of the program, and not to the password changing part. The criteria for comparison are the eight recommendations above. We shall not examine psychological acceptability, because that is more closely related to external influences such as the user's environment; we want to look at the proactive testing mechanisms only.

3.1. The 4.3 BSD UNIX System Password Changing Program *passwd*

This program [14] is similar to the password changing programs distributed by many vendors, so we use it as an example. Its proactive checking is confined to advising users that passwords should be at least 5 characters long, or at least 6 if they consist of monocase letters. The error message asks the user to use a longer password; but if the user insists (by entering the password 3 times), these requirements are waived.

This program fails to meet almost all of the requirements, except for number 7 and a very small part of number 2. The mechanism allows passwords that fail its tests to be used; it assumes the same criteria for good passwords hold across all users and sites; no aging is performed; as the tests are coded into the program, modifying them requires rewriting the program; and the tests cannot involve running the password through a subprogram. Requirement 7 (that all potential passwords not be listed in dictionaries) is met, because the test involves only character types and lengths; and as these are classes of passwords that are common, a small subset of requirement 2 is met.

It should be noted that on other UNIX systems (such as those based on System V) [13], password aging mechanisms are provided. However, the other requirements are not met.

3.2. An Obvious Password Detector

This program [7] works by detecting probable English words based on the number of occurrences of common triples. For example, in the word "hello", three triples occur – "hel", "ell", and "llo". Given a list of words, one can build a three-dimensional table in which all triples occurring in the words in the list are marked in the table. Other common triples, such as letters being repeated 3 times, are also entered. The triples are extracted from a proposed password and looked up; if most, or all, are found marked in the table, the password is an English word and is too easy to guess.

This software is distributed as a program into which the password is entered; output is a message saying the password is too obvious, too short, or acceptable. Because it is to be used with a password changer, meeting requirement 1 depends on how it is merged with that program.

This checker is a major improvement to the standard UNIX system password programs. It allows detection of common English words, and provides a mechanism for detecting strings that are too similar to English words. For example, it rejects “waters”, “watering”, and “bananna”, none of which are in the standard UNIX system dictionary but all of which would be considered “easily guessable”. More precisely, as distributed, the program does not distinguish between upper and lower case letters, and treats any non-letter as the same (non-alphabetic) character. The table was built from the system dictionary and a set of “obvious” patterns (such as “qwerty”). Hence of the 14 classes in table 1, it will reject words in at least 8, and possibly 10 (the uncertain classes being the transformations of user account and real names).

The requirement that the passwords to be rejected be entered in, or very similar to, words in the precompiled dictionaries is the major drawback of this scheme. For example, the string “Beijing” would be considered non-obvious to this scheme unless it had been entered in a dictionary. Further, common classes involving a transformation will often pass; for example, “bishop” is deemed too obvious, but “pohsib” is deemed acceptable. (The specific classes in table 1 that will be accepted are words with embedded digits, revered dictionary words, foreign words, or non-numeric license plates.) So the distributed table is clearly inadequate, and unless considerable care is spent in developing a new table, this scheme will not catch all of the classes of obvious passwords.

Requirements 3 and 4 are not met by the scheme, as distributed; however, with minimal extra work, the software could be altered to allow a separate table for each user, and one for the site, to be loaded. This speaks to the robustness of the concept.

Requirement 5 (aging) is not met, although by using an adaptive table (one into which the triples of past passwords were entered) it could be.

None of requirements 6 through 8 are met, because the tests are tied to the table. This program specifies two types of tests: is the word in a dictionary, or likely to be related to one in a dictionary; and is it too short (or too long)? The nature of the mechanism precludes being able to meet these requirements.

To be fair, we should note that the stated purpose of this proactive mechanism was to eliminate those passwords which were probably English words, and the algorithm chosen is suited to that purpose specifically. This package was not intended to be used as a general-purpose proactive mechanism.

3.3. The OPUS Project

This project [12] is similar to the Obvious Password Detector in purpose; the difference is that the implementation of OPUS uses multiple Bloom filters rather than triples.

When a word is given to OPUS, it is first transformed into a hashed form. This hash is then run through a series of filters each of which generates an integer output. The set of integers is then compared to a list of precomputed filter outputs; if the computed output is identical to any of the precomputed outputs, the word is rejected. The precomputed outputs are obtained by running words deemed unsuitable for passwords through the same filters. By varying the number of bits in the bit vectors, and the number of hash functions, one can control the probability of a false positive (that is, the number of proposed passwords rejected even though they are not in the dictionary).

OPUS and the Obvious Password Detector meet, and do not meet, the same requirements; they are simply different ways of detecting words in a dictionary. (The goal of both mechanisms is similar, OPUS being intended to catch obvious passwords which are not English words as well as English words.) Like the Obvious Password Detector, it has good potential provided it is made more flexible. But currently, unless considerable care is spent in developing the dictionary, OPUS may well miss some of the classes of obvious passwords, and unless the sets of filter outputs are changed on a per-user and per-site basis, it is too inflexible.

3.4. The Program *npasswd*

This program [3] allows several types of tests for easily guessed passwords. Users may specify a set of dictionaries to be checked and a minimum length requirement; they may also ban passwords with single case characters. The program itself will reject passwords consisting of repeated characters, illegal characters (which may be configured at run time), several transformations of the user's personal information (such as name or office building), and various per-site information such as the host name.

The functionality of this program is intended to be greater than that of the preceding two. It allows some per-user discrimination, and a great deal of per-site discrimination (in addition to building a site-specific dictionary, it will also test some system-specific strings). By building appropriate dictionaries, the administrator can configure it to reject any proposed password in a class of easily guessed passwords. Furthermore, setting up the configuration file is very simple: one simply lists the dictionaries to be checked, and adds appropriate keywords to perform the other tests (such as length). So requirements 1, 3, 4, 5 (when run on a UNIX system supporting password aging), and 6 are met. With respect to requirement 2, of the 14 classes in table 1, all can be detected with suitable dictionaries.

Requirements 7 and 8 are not met, and this is the major problem with *npasswd*: it does not allow for easy testing of variations of dictionary words. For example, "water" is in most system dictionaries, but "waters" is not. Hence, while "Water" would be rejected, "Waters" would not. While this could be alleviated by a larger dictionary, many sites will simply not have such a dictionary available. A similar comment holds for words consisting of digits only: as there is no pattern matching facility, any string of numbers of sufficient length will be accepted as a password unless all such numbers are entered into a dictionary. The failure to meet requirement 8 exacerbates this, as dictionaries must be stored either as word lists or as UNIX-style database files; they cannot be compressed without modifying the program.

3.5. The Program *passwd+ (beta)*

This program has been available since 1987 [2] in alpha test form; like the standard UNIX utility *passwd*, and the previously-mentioned *npasswd* program, it is both a password changer and a password checker. The password test mechanisms cannot be overridden or bypassed, so requirement 1 is met. It contains support for password aging when the underlying UNIX system supports it, so requirement 5 is met to a degree. The remaining requirements all relate to the nature of the testing, so let us focus on that.

What follows applies to the beta version, which has been recently completed. In addition to allowing dictionary tests, it allows tests to use pattern matching as well as to execute arbitrary programs and base rejection on the output. This enables it to reject passwords in any class that can be described using patterns or for which a program can be written to detect membership, thereby satisfying requirements 7 and 8; as for requirement 2, all 14 of the classes in table 1 can be described using either dictionaries, patterns, or programs. The configuration file uses variables which are set at each invocation, and include the user's name and host name (set automatically) and can include any information the site administrator wishes to provide. Hence it can perform both per-user and per-site discrimination, thereby satisfying requirements 3 and 4. Also, it can provide an aging facility in the same way that OPUS can be used to support aging: add the old password to a dictionary. This means requirement 5 can be met even if the underlying UNIX system does not support aging. Requirement 6, that setting up the tests be straightforward, is partially met because setting up tests merely requires the system administrator adding them to a configuration file.

The reservations implicit in the previous sentence arise from the structure of the little language used to express these tests. Because of the flexibility of *passwd+*'s configuration language, it allows more general proactive testing than the other four schemes. For example, one can send the hashed password over the network to a password cracker, which will then reply with "1" for "guessable" and "0" for "not guessable" (much as was done in CRACK [10]). But with this power comes complexity, and in some sense *passwd+* suffers from the inverse problem of *npasswd*: setting up the tests can be quite simple if the tests are simple, or quite complex if the tests are complex. So the interface for the system administrator can be difficult to use. (The users, of course, do not see this complexity.) On the other hand, when properly configured, it catches all passwords in the suspect class.

4. Comparison of the Programs

The programs described in the previous section fall into two classes: the *testing programs* which do not change passwords but merely test proposed passwords, and the *proactive password programs*, which change passwords as well as test proposed passwords. The two types are closely related, and two of the proactive password programs have separate password testing programs.

The two testing programs are both dictionary lookup programs. The Obvious Password Detector uses a statistical characteristic of the dictionary to eliminate words which are likely to be in the dictionary; OPUS uses Bloom filters. Both these techniques are probabilistic in nature and simply extend previous work (see for example [5][9]); similar techniques can be used with proactive password programs, and in fact either system could be incorporated into such a program.

Of the three proactive password programs described above, the standard UNIX program *passwd* is clearly the weakest; this is not surprising, given the philosophy of the UNIX operating system. That philosophy says to have one program to perform one task; and *passwd* was originally intended to change the user's password. As the UNIX operating system evolved, that philosophy was observed perhaps more in the breach than in the observance, and *passwd* was extended to change login shells, user information, to handle password aging, and extended in other ways. Unfortunately, rigorous password checking was not one of them; the precise reasons for this are not clear.

After the completion of a rather powerful password cracking program *deszip* [1], the author of that program decided to develop and distribute a countermeasure, so that sites which obtained *deszip* would be able to insulate their system against attacks from other password cracking programs. The original version of *passwd+* was distributed with *deszip* as well as separately; it provided all the testing facilities present in the beta version, including execution of subprograms and pattern matching tests as well as dictionary lookup tests. The complexity of the language used to express these tests, as well as some awkwardness about the configuration file and the lack of extensibility in other areas (for example, adding the ability to handle alternate password file formats was very difficult) has led to the work on a beta test version.

The program *npasswd* was developed independently and has been considerably more polished than *passwd+*. Although it does not provide as powerful a set of tests, the set it does provide are excellent, and configuration is very simple. For this reason, if a system administrator does not have the time to learn the little language of *passwd+* and is not happy with the sample configuration file, *npasswd* is often a better choice. On the other hand, *passwd+* clearly does provide the system administrator with a more comprehensive testing ability.

5. Conclusion

In this paper we analyzed some of the requirements of proactive password checkers, and compared several checkers which either are available or are under development. We tested each by scoring their abilities to meet specific criteria derived from the problems that proactive password checking is supposed to overcome. Our goal was to shed some light on the requirements the testing component of the proactive password checker should meet, and give examples of programs which attempted to implement some, or all of, the requirements.

6. References

[1] Bishop, M., "An Application of a Fast Data Encryption Standard Implementation," *Computing Systems* **1**(3) (Summer 1988) pp. 221-256.

[2] Bishop, M., "A Proactive Password Checker," in *Information Security: Proceedings of the IFIP TC11 Seventh International Conference on Information Security: Creating Confidence in Information Processing*, D. T. Lindsay and W. L. Price (eds.), North-Holland, New York, NY (1991) pp. 169-180.

[3] Hoover, C., *npasswd version 1.7* (Jan. 28, 1992). Available from ftp.cc.utexas.edu using anonymous ftp.

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

[5] McIlroy, M. D., "Development of a Spelling List," *IEEE Transactions on Communications* **COM-30**(1) (Jan. 1982) pp. 91-99.

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

[7] Nagle, J. B., "An Obvious Password Detector," *comp.sources.unix* **16**(60) (Nov. 1988). USENET message 1175@fig.bbn.com.

[8] *Password Management Guideline*, Department of Defense report CSC-STD-002-85 (Apr. 1985).

[9] Peterson, J. L., "Computer Programs for Detecting and Correcting Spelling Errors," *Communications of the ACM* **23**(12) (Dec. 1980) pp. 676-687.

[10] Raleigh, T. M. and Underwood, R. W., "CRACK: A Distributed Password Advisor," *Proceedings of the UNIX Security Workshop* (1988) pp. 12-13.

[11] Saltzer, J. H. and Schroeder, M. D., "The Protection of Information in Computer Systems," *Proceedings of the IEEE* **63**(9) (Sep. 1975) pp. 1278-1308.

[12] Spafford, E., "Preventing Weak Password Choices," *Proceedings of the Fourteenth National Computer Security Conference* (Oct. 1991) pp. 446-455.

[13] *System V Interface Definition*, *Volume II*, AT&T (1986).

[14] *UNIX User's Reference Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (April 1986). Reprinted by the USENIX Association (June 1987).