"Foiling the Cracker": A Survey of, and Improvements to, Password Security†

Daniel V. Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15217
dvk@sei.cmu.edu
+1 412 268 7791

ABSTRACT

With the rapid burgeoning of national and international networks, the question of system security has become one of growing importance. High speed inter-machine communication and even higher speed computational processors have made the threats of system "crackers," data theft, data corruption very real. This paper outlines some of the problems of current password security by demonstrating the ease by which individual accounts may be broken. Various techniques used by crackers are outlined, and finally one solution to this point of system vulnerability, a proactive password checker, is proposed.

1. Introduction

The security of accounts and passwords has always been a concern for the developers and users of Unix. When Unix was younger, the password encryption algorithm was a simulation of the M-209 cipher machine used by the U.S. Army during World War II. [Morris1979] This was a fair encryption mechanism in that it was difficult to invert under the proper circumstances, but suffered in that it was too fast an algorithm. On a PDP-11/70, each encryption took approximately 1.25ms, so that it was possible to check roughly 800 passwords/second. Armed with a dictionary of 250,000 words, a cracker could compare their encryptions with those all stored in the password file in a little more than five minutes. Clearly, this was a security hole worth filling.

In later (post-1976) versions of Unix, the DES algorithm [DES1975] was used to encrypt passwords. The user's password is used as the DES key, and the algorithm is used to encrypt a constant. The algorithm is iterated 25 times, with the result being an 11 character string plus a 2-character "salt." This method is similarly difficult to decrypt (further complicated through the introduction of one of 4096 possible salt values) and had the added advantage of being slow. On a μVAX-II (a machine substantially faster than a PDP-11/70), a single encryption takes on the order of 280ms, so that a determined cracker can only check approximately 3.6 encryptions a second. Checking this same dictionary of 250,000 words would now take over 19 *hours* of CPU time. Although this is still not very much time to break a single account, there is no guarantee that this account will use one of these words as a password. Checking the passwords on a system with 50 accounts would take on average 40 CPU *days* (since the random selection of salt values practically guarantees that each user's password will be encrypted with a different salt), with no guarantee of success. If this new, slow algorithm was combined with the user education needed to prevent the selection of obvious passwords, the problem seemed solved.

Regrettably, two recent developments and the recurrence of an old one have brought the problem of password security back to the fore.

† This work was sponsored in part by the U.S. Department of Defense.

1) CPU speeds have gotten increasingly faster since 1976, so much so that processors that are 25-40 times faster than the PDP-11/70 (e.g., the DECstation 3100 used in this research) are readily available as desktop workstations. With inter-networking, many sites have hundreds of the individual workstations connected together, and enterprising crackers are discovering that the ‘‘divide and conquer’’ algorithm can be extended to multiple processors, especially at night when those processors are not otherwise being used. Literally thousands of times the computational power of 10 years ago can be used to break passwords.

2) New implementations of the DES encryption algorithm have been developed, so that the time it takes to encrypt a password and compare the encryption against the value stored in the password file has dropped below the 1ms mark. [Bishop1988, Feldmeier1989] On a single workstation, the dictionary of 250,000 words can once again be cracked in under five minutes. By dividing the work across multiple workstations, the time required to encrypt these words against all 4096 salt values could be no more than an hour or so. With a recently described hardware implementation of the DES algorithm, the time for each encryption can be reduced to approximately 6 μs. [Leong1991] This means that this same dictionary can be be cracked in only 1.5 seconds.

3) Users are rarely, if ever, educated as to what are wise choices for passwords. If a password is in a dictionary, it is extremely vulnerable to being cracked, and users are simply not coached as to ‘‘safe’’ choices for passwords. Of those users who are so educated, many think that simply because their password is not in */usr/dict/words*, it is safe from detection. Many users also say that because they do not have any private files on-line, they are not concerned with the security of their account, little realizing that by providing an entry point to the system they allow damage to be wrought on their entire system by a malicious cracker.

Because the entirety of the password file is readable by all users, the encrypted passwords are vulnerable to cracking, both on-site and off-site. Many sites have responded to this threat with a reactive solution − they scan their own password files and advise those users whose passwords they are able to crack. The problem with this solution is that while the local site is testing its security, the password file is still vulnerable from the outside. The other problems, of course, are that the testing is very time consuming and only reports on those passwords it is able to crack. It does nothing to address user passwords which fall outside of the specific test cases (e.g., it is possible for a user to use as a password the letters ‘‘qwerty’’ − if this combination is not in the in-house test dictionary, it will not be detected, but there is nothing to stop an outside cracker from having a more sophisticated dictionary!).

Clearly, one solution to this is to either make */etc/passwd* unreadable, or to make the encrypted password portion of the file unreadable. Splitting the file into two pieces − a readable */etc/passwd* with all but the encrypted password present, and a ‘‘shadow password’’ file that is only readable by **root** is the solution proposed by Sun Microsystems (and others) that appears to be gaining popularity. It seems, however, that this solution will not reach the majority of non-Sun systems for quite a while, nor even, in fact, many Sun systems, due to many sites’ reluctance to install new releases of software.†

What I propose, therefore, is a publicly available *proactive* password checker, which will enable users to change their passwords, and to check *a priori* whether the new password is ‘‘safe.’’ The criteria for safety should be tunable on a per-site basis, depending on the degree of security desired. For example, it should be possible to specify a minimum length password, a restriction that only lower case letters are not allowed, that a password that looks like a license plate be illegal, and so on. Because this proactive checker will deal with the pre-encrypted passwords, it will be able to perform more sophisticated pattern matching on the password, and will be able to test the safety without having to go through the effort of cracking the encrypted version. Because the checking will be done automatically, the process of education can be transferred to the machine, which will instruct the user *why* a particular choice of password is bad.

† The problem of lack of password security is not just endemic to Unix. A recent Vax/VMS worm had great success by simply trying the username as the password. Even though the VMS user authorization file is inaccessible to ordinary users, the cracker simply tried a number of ‘‘obvious’’ password choices − and easily gained access.

2. Password Vulnerability

It has long been known that all a cracker need do to acquire access to a Unix machine is to follow two simple steps, namely:

1) Acquire a copy of that site's */etc/passwd* file, either through an unprotected *uucp* link, well known holes in *sendmail*, or via *ftp* or *tftp*.
2) Apply the standard (or a sped-up) version of the password encryption algorithm to a collection of words, typically */usr/dict/words* plus some permutations on account and user names, and compare the encrypted results to those found in the purloined */etc/passwd* file.

If a match is found (and often at least one will be found), the cracker has access to the targeted machine. Certainly, this mode of attack has been known for some time [Spafford1988] and the defenses against this attack have also long been known. What is lacking from the literature is an accounting of just how vulnerable sites are to this mode of attack. In short, many people know that there is a problem, but few people believe it applies to them.

"There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys." [Grampp1984] The problem here, therefore, is how to divulge useful information on the vulnerability of systems, without providing too much information, since almost certainly this information could be used by a cracker to break into some as-yet unviolated system. Most of the work that I did was of a general nature − I did not focus on a particular user or a particular system, and I did not use any personal information that might be at the disposal of a dedicated "bad guy." Thus any results which I have been able to garner indicate only general trends in password usage, and cannot be used to great advantage when breaking into a particular system. This generality notwithstanding, I am sure that any self-respecting cracker would already have these techniques at their disposal, and so I am not bringing to light any great secret. Rather, I hope to provide a basis for protection for systems that can guard against future attempts at system invasion.

2.1. The Survey and Initial Results

In October and again in December of 1989, I asked a number of friends and acquaintances around the United States and Great Britain to participate in a survey. Essentially what I asked them to do was to mail me a copy of their */etc/passwd* file, and I would try to crack their passwords (and as a side benefit, I would send them a report of the vulnerability of their system, although at no time would I reveal individual passwords nor even of their sites participation in this study). Not surprisingly, due to the sensitive nature of this type of disclosure, I only received a small fraction of the replies I hoped to get, but was nonetheless able to acquire a database of nearly 15,000 account entries. This, I hoped, would provide a representative cross section of the passwords used by users in the community.

Each of the account entries was tested by a number of intrusion strategies, which will be covered in greater detail in the following section. The possible passwords that were tried were based on the user's name or account number, taken from numerous dictionaries (including some containing foreign words, phrases, patterns of keys on the keyboard, and enumerations), and from permutations and combinations of words in those dictionaries. All in all, after nearly 12 CPU months of rather exhaustive testing, approximately 25% of the passwords had been guessed. So that you do not develop a false sense of security too early, I add that 21% (nearly 3,000 passwords) were guessed in the first week, and that in the first 15 minutes of testing, 368 passwords (or 2.7%) had been cracked using what experience has shown would be the most fruitful line of attack (i.e., using the user or account names as passwords). These statistics are frightening, and well they should be. On an average system with 50 accounts in the */etc/passwd* file, one could expect the first account to be cracked in under 2 minutes, with 5−15 accounts being cracked by the end of the first day. Even though the **root** account may not be cracked, all it takes is one account being compromised for a cracker to establish a toehold in a system. Once that is done, any of a number of other well-known security loopholes (many of which have been published on the network) can be used to access or destroy any information on the machine.

It should be noted that the results of this testing do not give us any indication as to what the *uncracked* passwords are. Rather, it only tells us what was essentially already known − that users are likely to use words that are familiar to them as their passwords. [Riddle1989] What new information it did provide, however, was the *degree* of vulnerability of the systems in question, as well as providing a basis for developing a proactive password changer – a system which pre-checks a password before it is entered into the system, to determine whether that password will be vulnerable to this type of attack. Passwords which can be derived from a dictionary are clearly a bad idea [Alvare1988] and users should be prevented from using them. Of course, as part of this censoring process, users should also be told *why* their proposed password is not good, and what a good class of password would be.

As to those passwords which remain unbroken, I can only conclude that these are much more secure and ‘‘safe’’ than those to be found in my dictionaries. One such class of passwords is word pairs, where a password consists of two short words, separated by a punctuation character. Even if only words of 3 to 5 lower case characters are considered, */usr/dict/words* provides 3000 words for pairing. When a single intermediary punctuation character is introduced, the sample size of 90,000,000 possible passwords is rather daunting. On a DECstation 3100, testing each of these passwords against that of a single user would require over 25 CPU *hours* – and even then, no guarantee exists that this is the type of password the user chose. Introducing one or two upper case characters into the password raises the search set size to such magnitude as to make cracking untenable.

Another ‘‘safe’’ password is one constructed from the initial letters of an easily remembered, but not too common phrase. For example, the phrase ‘‘Unix is a trademark of Bell Laboratories’’ could give rise to the password ‘‘UiatoBL.’’ This essentially creates a password which is a random string of upper and lower case letters. Exhaustively searching this list at 1000 tests per second with only 6 character passwords would take nearly 230 CPU days. Increasing the phrase size to 7 character passwords makes the testing time over 32 CPU *years* – a Herculean task that even the most dedicated cracker with huge computational resources would shy away from.

Thus, although I don’t know what passwords were chosen by those users I was unable to crack, I can say with some surety that it is doubtful that anyone else could crack them in a reasonable amount of time, either.

2.2. Method of Attack

A number of techniques were used on the accounts in order to determine if the passwords used for them were able to be compromised. To speed up testing, all passwords with the same salt value were grouped together. This way, one encryption per password per salt value could be performed, with multiple string comparisons to test for matches. Rather than considering 15,000 accounts, the problem was reduced to 4,000 salt values. The password tests were as follows:

1) Try using the user’s name, initials, account name, and other relevant personal information as a possible password. All in all, up to 130 different passwords were tried based on this information. For an account name **klone** with a user named ‘‘Daniel V. Klein,’’ some of the passwords that would be tried were: klone, klone0, klone1, klone123, dvk, dvkdvk, dklein, DKlein, leinad, nielk, dvklein, danielk, DvkkvD, DANIEL-KLEIN, (klone), KleinD, etc.

2) Try using words from various dictionaries. These included lists of men’s and women’s names (some 16,000 in all); places (including permutations so that ‘‘spain,’’ ‘‘spanish,’’ and ‘‘spaniard’’ would all be considered); names of famous people; cartoons and cartoon characters; titles, characters, and locations from films and science fiction stories; mythical creatures (garnered from Bulfinch’s mythology and dictionaries of mythical beasts); sports (including team names, nicknames, and specialized terms); numbers (both as numerals – ‘‘2001,’’ and written out – ‘‘twelve’’); strings of letters and numbers (‘‘a,’’ ‘‘aa,’’ ‘‘aaa,’’ ‘‘aaaa,’’ etc.); Chinese syllables (from the Pinyin Romanization of Chinese, a international standard system of writing Chinese on an English keyboard); the King James Bible; biological terms; common and vulgar phrases (such as ‘‘fuckyou,’’ ‘‘ibmsux,’’ and ‘‘deadhead’’); keyboard patterns (such as ‘‘qwerty,’’ ‘‘asdf,’’ and ‘‘zxcvbn’’); abbreviations (such as ‘‘roygbiv’’ – the colors in the rainbow, and ‘‘ooottafagvah’’ – a mnemonic for remembering the 12 cranial nerves); machine names (acquired from */etc/hosts*); characters, plays, and locations from Shakespeare; common Yiddish words; the names of asteroids; and a collection of words from various technical papers I had previously published. All told, more than 60,000 separate words were considered per user (with any inter- and intra-dictionary duplicates being discarded).

3) Try various permutations on the words from step 2. This included making the first letter upper case or a control character, making the entire word upper case, reversing the word (with and without the aforementioned capitalization), changing the letter ‘o’ to the digit ‘0’ (so that the word “scholar” would also be checked as “sch0lar”), changing the letter ‘l’ to the digit ‘1’ (so that “scholar” would also be checked as “scho1ar,” and also as “sch01ar”), and performing similar manipulations to change the letter ‘z’ into the digit ‘2’, and the letter ‘s’ into the digit ‘5’. Another test was to make the word into a plural (irrespective of whether the word was actually a noun), with enough intelligence built in so that “dress” became “dresses,” “house” became “houses,” and “daisy” became “daisies.” We did not consider pluralization rules exhaustively, though, so that “datum” forgivably became “datums” (not “data”), while “sphynx” became “sphynxs” (and not “sphynges”). Similarly, the suffixes “-ed,” “-er,” and “-ing” were added to transform words like “phase” into “phased,” “phaser,” and “phasing.” These 14 to 17 additional tests per word added another 1,000,000 words to the list of possible passwords that were tested for each user.

4) Try various capitalization permutations on the words from step 2 that were not considered in step 3. This included all single letter capitalization permutations (so that “michael” would also be checked as “mIchael,” “miChael,” “micHael,” “michAel,” etc.), double letter capitalization permutations (“MIchael,” “MiChael,” “MicHael,” ... , “mIChael,” “mIcHael,” etc.), triple letter permutations, and so on. The single letter permutations added roughly another 400,000 words to be checked per user, while the double letter permutations added another 1,500,000 words. Three letter permutations would have added at least another 3,000,000 words *per user* had there been enough time to complete the tests. Tests of 4, 5, and 6 letter permutations were deemed to be impracticable without much more computational horsepower to carry them out.

5) Try foreign language words on foreign users. The specific test that was performed was to try Chinese language passwords on users with Chinese names. The Pinyin Romanization of Chinese syllables was used, combining syllables together into one, two, and three syllable words. Because no tests were done to determine whether the words actually made sense, an exhaustive search was initiated. Since there are 398 Chinese syllables in the Pinyin system, there are 158,404 two syllable words, and slightly more than 16,000,000 three syllable words.† A similar mode of attack could as easily be used with English, using rules for building pronounceable nonsense words.

6) Try word pairs. The magnitude of an exhaustive test of this nature is staggering. To simplify this test, only words of 3 or 4 characters in length from */usr/dict/words* were used. Even so, the number of word pairs is $\mathbf{O}(10^7)$ (multiplied by 4096 possible salt values), and as of this writing, the test is only 10% complete.

For this study, I had access to four DECstation 3100’s, each of which was capable of checking approximately 750 passwords per second. Even with this total peak processing horsepower of 3,000 tests per second (some machines were only intermittently available), testing the $\mathbf{O}(10^{10})$ password/salt pairs for the first four tests required on the order of 12 CPU *months* of computations. The remaining two tests are still ongoing after an additional 18 CPU months of computation. Although for research purposes this is well within acceptable ranges, it is a bit out of line for any but the most dedicated and resource-rich cracker.

2.3. Summary of Results

The problem with using passwords that are derived directly from obvious words is that when a user thinks “Hah, no one will guess this permutation,” they are almost invariably wrong. Who would ever suspect that I would find their passwords when they chose “fylgjas” (guardian creatures from Norse mythology), or the Chinese word for “hen-pecked husband”? No matter what words or permutations thereon are chosen for a password, if they exist in some dictionary, they are susceptible to directed cracking. The following table

---

† The astute reader will notice that 398^3 is in fact 63,044,972. Since Unix passwords are truncated after 8 characters, however, the number of unique polysyllabic Chinese passwords is only around 16,000,000. Even this reduced set was too large to complete under the imposed time constraints.

give an overview of the types of passwords which were found through this research.

A note on the table is in order. The number of matches given from a particular dictionary is the total number of matches, irrespective of the permutations that a user may have applied to it. Thus, if the word ‘‘wombat’’ were a particularly popular password from the biology dictionary, the following table will not indicate whether it was entered as ‘‘wombat,’’ ‘‘Wombat,’’ ‘‘TABMOW,’’ ‘‘w0mbat,’’ or any of the other 71 possible differences that this research checked. In this way, detailed information can be divulged without providing much knowledge to potential ‘‘bad guys.’’

Additionally, in order to reduce the total search time that was needed for this research, the checking program eliminated both inter- and intra-dictionary duplicate words. The dictionaries are listed in the order tested, and the total size of the dictionary is given in addition to the number of words that were eliminated due to duplication. For example, the word ‘‘georgia’’ is both a female name and a place, and is only considered once. A password which is identified as being found in the common names dictionary might very well appear in other dictionaries. Additionally, although ‘‘duplicate,’’ ‘‘duplicated,’’ ‘‘duplicating’’ and ‘‘duplicative’’ are all distinct words, only the first eight characters of a password are used in Unix, so all but the first word are discarded as redundant.

Passwords cracked from a sample set of 13,797 accounts

Type of Password	Size of Dictionary	Duplicates Eliminated	Search Size	# of Matches	Pct. of Total	Cost/Benefit Ratio*
User/account name	130†	–	130	368	2.7%	2.830
Character sequences	866	0	866	22	0.2%	0.025
Numbers	450	23	427	9	0.1%	0.021
Chinese	398	6	392	56	0.4%‡	0.143
Place names	665	37	628	82	0.6%	0.131
Common names	2268	29	2239	548	4.0%	0.245
Female names	4955	675	4280	161	1.2%	0.038
Male names	3901	1035	2866	140	1.0%	0.049
Uncommon names	5559	604	4955	130	0.9%	0.026
Myths & legends	1357	111	1246	66	0.5%	0.053
Shakespearean	650	177	473	11	0.1%	0.023
Sports terms	247	9	238	32	0.2%	0.134
Science fiction	772	81	691	59	0.4%	0.085
Movies and actors	118	19	99	12	0.1%	0.121
Cartoons	133	41	92	9	0.1%	0.098
Famous people	509	219	290	55	0.4%	0.190
Phrases and patterns	998	65	933	253	1.8%	0.271
Surnames	160	127	33	9	0.1%	0.273
Biology	59	1	58	1	0.0%	0.017
/usr/dict/words	24474	4791	19683	1027	7.4%	0.052
Machine names	12983	3965	9018	132	1.0%	0.015
Mnemonics	14	0	14	2	0.0%	0.143
King James bible	13062	5537	7525	83	0.6%	0.011
Miscellaneous words	8146	4934	3212	54	0.4%	0.017
Yiddish words	69	13	56	0	0.0%	0.000
Asteroids	3459	1052	2407	19	0.1%	0.007
Total	86280	23553	62727	**3340**	**24.2%**	0.053

* In all cases, the cost/benefit ratio is the number of matches divided by the search size. The more words that needed to be tested for a match, the lower the cost/benefit ratio.

† The dictionary used for user/account name checks naturally changed for each user. Up to 130 different permutations were tried for each.

‡ While monosyllablic Chinese passwords were tried for all users (with 12 matches), polysyllabic Chinese passwords were tried only for users with Chinese names. The percentage of matches for this subset of users is 8% – a greater hit ratio

The results are quite disheartening. The total size of the dictionary was only 62,727 words (not counting various permutations). This is much smaller than the 250,000 word dictionary postulated at the beginning of this paper, yet armed even with this small dictionary, nearly 25% of the passwords were cracked!

Length of Cracked Passwords

Length	Count	Percentage
1 character	4	0.1%
2 characters	5	0.2%
3 characters	66	2.0%
4 characters	188	5.7%
5 characters	317	9.5%
6 characters	1160	34.7%
7 characters	813	24.4%
8 characters	780	23.4%

The results of the word-pair tests are not included in either of the two tables. However, at the time of this writing, the test was approximately 10% completed, having found an additional 0.4% of the passwords in the sample set. It is probably reasonable to guess that a total of 4% of the passwords would be cracked by using word pairs.

3. Action, Reaction, and Proaction

What then, are we to do with the results presented in this paper? Clearly, something needs to be done to safeguard the security of our systems from attack. It was with intention of enhancing security that this study was undertaken. By knowing what kind of passwords users use, we are able to prevent them from using those that are easily guessable (and thus thwart the cracker).

One approach to eliminating easy-to-guess passwords is to periodically run a password checker – a program which scans */etc/passwd* and tries to break the passwords in it. [Raleigh1988] This approach has two major drawbacks. The first is that the checking is very time consuming. Even a system with only 100 accounts can take over a month to diligently check. A halfhearted check is almost as bad as no check at all, since users will find it easy to circumvent the easy checks and still have vulnerable passwords. The second drawback is that it is very resource consuming. The machine which is being used for password checking is not likely to be very useful for much else, since a fast password checker is also extremely CPU intensive.

Another popular approach to eradicating easy-to-guess passwords is to force users to change their passwords with some frequency. In theory, while this does not actually eliminate any easy-to-guess passwords, it prevents the cracker from dissecting */etc/passwd* "at leisure," since once an account is broken, it is likely that that account will have had it's password changed. This is of course, only theory. The biggest disadvantage is that there is usually nothing to prevent a user from changing their password from "Daniel" to "Victor" to "Klein" and back again (to use myself as an example) each time the system demands a new password. Experience has shown that even when this type of password cycling is precluded, users are easily able to circumvent simple tests by using easily remembered (and easily guessed) passwords such as "dvkJanuary," "dvkFebruary," etc. [Reid1989] A good password is one that is easily remembered, yet difficult to guess. When confronted with a choice between remembering a password or creating one that is hard to guess, users will almost always opt for the easy way out, and throw security to the wind.

Which brings us to the third popular option, namely that of assigned passwords. These are often words from a dictionary, pronounceable nonsense words, or random strings of characters. The problems here are numerous and manifest. Words from a dictionary are easily guessed, as we have seen. Pronounceable nonsense words (such as "trobacar" or "myclepate") are often difficult to remember, and random strings of characters (such as "h3rT+aQz") are even harder to commit to memory. Because these passwords have no personal mnemonic association to the users, they will often write them down to aid in their recollection. This immediately discards any security that might exist, because now the password is visibly associated with the system in question. It is akin to leaving the key under the door mat, or writing the combination to

than any other method. Because the dictionary size is over 16×10^6, though, the cost/benefit ratio is infinitesimal.

a safe behind the picture that hides it.

A fourth method is the use of ‘‘smart cards.’’ These credit card sized devices contain some form of encryption firmware which will ‘‘respond’’ to an electronic ‘‘challenge’’ issued by the system onto which the user is attempting to gain acccess. Without the smart card, the user (or cracker) is unable to respond to the challenge, and is denied access to the system. The problems with smart cards have nothing to do with security, for in fact they are very good warders for your system. The drawbacks are that they can be expensive and must be carried at all times that access to the system is desired. They are also a bit of overkill for research or educational systems, or systems with a high degree of user turnover.

Clearly, then, since all of these systems have drawbacks in some environments, an additional way must be found to aid in password security.

3.1. A Proactive Password Checker

The best solution to the problem of having easily guessed passwords on a system is to prevent them from getting on the system in the first place. If a program such as a password checker *reacts* by detecting guessable passwords already in place, then although the security hole is found, the hole existed for as long as it took the program to detect it (and for the user to again change the password). If, however, the program which changes user's passwords (i.e., */bin/passwd*) checks for the safety and guessability *before* that password is associated with the user's account, then the security hole is never put in place.

In an ideal world, the proactive password changer would require eight character passwords which are not in any dictionary, with at least one control character or punctuation character, and mixed upper and lower case letters. Such a degree of security (and of accompanying inconvenience to the users) might be too much for some sites, though. Therefore, the proactive checker should be tuneable on a per-site basis. This tuning could be accomplished either through recompilation of the *passwd* program, or more preferably, through a site configuration file.

As distributed, the behavior of the proactive checker should be that of attaining maximum password security – with the system administrator being able to turn off certain checks. It would be desireable to be able to test for and reject all password permutations that were detected in this research (and others), including:

- Passwords based on the user's account name
- Passwords based on the user's initials or given name
- Passwords which exactly match a word in a dictionary (not just */usr/dict/words*)
- Passwords which match a word in the dictionary with some or all letters capitalized
- Passwords which match a reversed word in the dictionary
- Passwords which match a reversed word in the dictionary with some or all letters capitalized
- Passwords which match a word in a dictionary with an arbitrary letter turned into a control character
- Passwords which match a dictionary word with the numbers ‘0’, ‘1’, ‘2’, and ‘5’ substituted for the letters ‘o’, ’l’, ’z’, and ’s’
- Passwords which are simple conjugations of a dictionary word (i.e., plurals, adding ‘‘ing’’ or ‘‘ed’’ to the end of the word, etc.)
- Passwords which are patterns from the keyboard (i.e., ‘‘aaaaaa’’ or ‘‘qwerty’’)
- Passwords which are shorter than a specific length (i.e., nothing shorter than six characters)
- Passwords which consist solely of numeric characters (i.e., Social Security numbers, telephone numbers, house addresses or office numbers)
- Passwords which do not contain mixed upper and lower case, or mixed letters and numbers, or mixed letters and punctuation
- Passwords which look like a state-issued license plate number

The configuration file which specifies the level of checking need not be readable by users. In fact, making this file unreadable by users (and by potential crackers) enhances system security by hiding a valuable guide to what passwords *are* acceptable (and conversely, which kind of passwords simply cannot be found).

Of course, to make this proactive checker more effective, it woule be necessary to provide the dictionaries that were used in this research (perhaps augmented on a per-site basis). Even more importantly, in addition to rejecting passwords which could be easily guessed, the proactive password changer would also have to tell the user *why* a particular password was unacceptable, and give the user suggestions as to what an acceptable password looks like.

4. Conclusion (and Sermon)

It has often been said that ‘‘good fences make good neighbors.’’ On a Unix system, many users also say that ‘‘I don’t care who reads my files, so I don’t need a good password.’’ Regrettably, leaving an account vulnerable to attack is not the same thing as leaving files unprotected. In the latter case, all that is at risk is the data contained in the unprotected files, while in the former, the whole system is at risk. Leaving the front door to your house open, or even putting a flimsy lock on it, is an invitation to the unfortunately ubiquitous people with poor morals. The same holds true for an account that is vulnerable to attack by password cracking techniques.

While it may not be actually true that good fences make good neighbors, a good fence at least helps keep out the bad neighbors. Good passwords are equivalent to those good fences, and a proactive checker is one way to ensure that those fences are in place *before* a breakin problem occurs.

References

%A Robert T. Morris %A Ken Thompson %T Password Security: A Case History %J Communications of the ACM %V 22 %N 11 %P 594-597 %D November 1979 %L Morris1979 .ds [F DES1975 %T Proposed Federal Information Processing Data Encryption Standard %J Federal Register (40FR12134) %D March 17, 1975 %L DES1975 .ds [F Bishop1988 %A Matt Bishop %T An Application of a Fast Data Encryption Standard Implementation %J Computing Systems %V 1 %N 3 %P 221-254 %D Summer 1988 %L Bishop1988 .ds [F Feldmeier1989 %A David C. Feldmeier %A Philip R. Karn %T UNIX Password Security – Ten Years Later %J CRYPTO Proceedings %D Summer 1989 %L Feldmeier1989 .ds [F Leong1991 %A Philip Leong %A Chris Tham %T UNIX Password Encryption Considered Insecure %J USENIX Winter Conference Proceedings %D January 1991 %L Leong1991 .ds [F Spafford1988 %A Eugene H. Spafford %T The Internet Worm Program: An Analysis %R Purdue Technical Report CSD-TR-823 %I Purdue University %D November 29, 1988 %L Spafford1988 .ds [F Grampp1984 %A F. Grampp %A R. Morris %T Unix Operating System Security %J AT&T Bell Labs Technical Journal %V 63 %N 8 %P 1649-1672 %D October 1984 %L Grampp1984 .ds [F Riddle1989 %A Bruce L. Riddle %A Murray S. Miron %A Judith A. Semo %T Passwords in Use in a University Timesharing Environment %J Computers & Security %V 8 %N 7 %P 569-579 %D November 1989 %L Riddle1989 .ds [F Alvare1988 %A Ana Marie De Alvare %A E. Eugene Schultz, Jr. %T A Framework for Password Selection %J USENIX UNIX Security Workshop Proceedings %D August 1988 %L Alvare1988 .ds [F Raleigh1988 %A T. Raleigh %A R. Underwood %T CRACK: A Distributed Password Advisor %J USENIX UNIX Security Workshop Proceedings %D August 1988 %L Raleigh1988 .ds [F Reid1989 %A Dr. Brian K Reid %D 1989 %I DEC Western Research Laboratory %O Personal communication. %L Reid1989 .nr [W 60550