Improving System Security via Proactive Password Checking

Matt Bishop‡

Department of Computer Science
University of California at Davis
Davis, CA 95616-8562
bishop@cs.ucdavis.edu
+1 916 752 8060

Daniel V. Klein

LoneWolf Systems, Inc.
133 Lanford Drive
Pittsburgh, PA 15235-1856
dvk@lonewolf.com
+1 412 242 5245

ABSTRACT

As the Internet has grown, its user community has changed from a small tight knit group of researchers to a loose gathering of people on a global network. The amazing and constantly growing numbers of machines and users ensures that untrustworthy individuals have full access to that network. 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 documented.

1. Introduction

The security of accounts and passwords has always been a concern for the developers and users of UNIX† systems. 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 [1]. 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 for a password cracker 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.

‡ Supported in part by grant NAG2-698 from the National Aeronautics and Space Administration to Dartmouth College. Portions of this work were done while the author was at Dartmouth College.

† UNIX is a trademark of Bell Laboratories.

Appeared in *Computers and Security* **14**(3) pp. 233-249 (1995)

Clearly, this was a security hole worth filling.

In later (post-1976) versions of UNIX, the DES algorithm [2] was used to encrypt passwords. The user's password is used as the DES key, and the algorithm is used to encrypt a constant (usually a string of nulls). The algorithm is iterated 25 times, with the result being an 11 character string plus a 2-character "salt". This method is very difficult to reverse (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 took on the order of 280ms, so that a determined cracker could only check approximately 3.6 encryptions a second. Checking this same dictionary of 250,000 words now took over 19 *hours* of CPU time. Although this is still not very much time to break a single account, there was no guarantee that this account would 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.

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

1) CPU speeds today are substantially faster than in 1976, so much so that readily obtainable and easily affordable processors are 25-100 times faster than the VAXen targeted by the "new" password encryptions. The DECstation 3100 and Sparc 1 used in the password cracking research were considered very fast machines 5 years ago. They have, like the tortoise of fable, been sped past with newer machines that are more than 10 times their speed. With internetworking, many sites have hundreds of 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 [3, 4]. On a single workstation, the dictionary of 250,000 words can once again be cracked in well 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 [5]. This means that this same dictionary could be cracked in only 1.5 seconds.

3) Users are rarely 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!).

† 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 to numerous machines.

Clearly, one solution to this is to either make */etc/passwd* unreadable (a simple solution which nonetheless breaks many legitimate tools), 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 this paper proposes, therefore, is a *proactive* password checker, which will enable users to change their passwords, and to check *a priori* whether the new password is ‘‘safe’’ from cracking. The criteria for safety are tunable on a per-site basis, depending on the degree of security desired. For example, it is 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 deals with the passwords in the clear (that is, before they are encrypted), it is able to perform more sophisticated pattern matching on the password, and is able to test the safety of a password without having to go through the effort of cracking the encrypted version. Because the checking is done automatically every time a user attempts to change his or her password, the process of education can be transferred to the machine, which will instruct the user *why* a particular choice of password is bad.

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:

1) Acquire a copy of that site’s */etc/passwd* file, either through an unprotected *uucp* link, well known holes in *sendmail*, via *ftp* or *tftp*, or other overt and covert means.

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 usually *at least* one will be) [6], the cracker has access to the targeted machine.

This mode of attack has been known for some time [Morris1979] [7], and the defenses against this attack have also long been known. How well sites protect themselves from the various modes of attack varies greatly from site to site. The publicly available proactive checker described in this paper will enable sites to protect themselves from a variety of attacks by providing a single "silver bullet" to address the many vulnerabilities.

2.1. The Survey and Initial Results

In late 1989, a number of site administrators cooperated in a study in password security. They submitted their copies of */etc/passwd* for cracking, yielding a total of nearly 14,000 account entries. Each of the entries was tested by a number of guessing strategies – the possible passwords that were tested 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.

After over 3 CPU years of rather exhaustive testing, approximately 40% of the passwords had been guessed. This represents the combined computing horsepower of 35 Sparc-2 workstations operating in parallel. In the first week, approximately 21% (nearly 3,000 passwords) were guessed using a *single* DECStation 3100 workstation; in fact, in the very first 15 minutes of testing, 458 passwords (or 3.2%) 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). All told, 30 root accounts were compromised. 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 [8]. 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 [9], 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.

2.2. Passwords to Avoid

A number of techniques were used on the accounts in order to determine if the passwords used for them were able to be compromised. Because any self respecting cracker would also try these tests, they should be checked in a proactive password changer. The password cracking tests were as follows:

1) The user's name, initials, account name, and other relevant personal information. 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, enolk, ENOLK, KleinD, etc.

2) Words from various dictionaries. For our research, these included lists of first and last names names (some 35,000 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, science fiction stories and Shakespeare; 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," "abab," etc.); 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*); common Yiddish words; the names of asteroids; a collection of words from various

technical papers, recipes, and scripts; foreign language dictionaries (including Chinese, Dutch, French, German, Greek, Italian, Norwegian, and Swedish). All told, more than 650,000 separate words were considered per user (with any inter- and intra-dictionary duplicates being discarded). All these dictionaries are publicly available from various sites across the Internet.

3) Various permutations on the words from step 2. These 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), capitalizing random letters, 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 and add the suffixes ‘‘-ed,’’ ‘‘-er,’’ and ‘‘-ing’’ to transform words like ‘‘phase’’ into ‘‘phases,’’ ‘‘phased,’’ ‘‘phaser,’’ and ‘‘phasing.’’

4) 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 examined. Even so, the number of word pairs is $\mathbf{O}(10^7)$ (multiplied by 4096 possible salt values), but despite this magnitude, this line of attack was surprisingly fruitful.

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 we would find their passwords when they chose ‘‘fylgjas’’ (guardian creatures from Norse mythology), or ‘‘pataitai’’ (the Chinese word for ‘‘hen-pecked husband’’)? No matter what words or permutations thereon are chosen for a password, if they exist in *any* on-line dictionary, they are susceptible to directed cracking. The following two tables give an overview of the types of passwords which were found (out of a sample set of 13,892 accounts) through this research.

Distribution of Cracked Passwords by Type		
Type of Password	**Number Successfully Cracked**	**Percentage**
Account/User Name	458	3.2%
Numbers	100	0.7%
Character Combinations	93	0.6%
Names	1003	7.2%
Permuted Names	195	1.4%
Words	2128	15.3%
Permuted Words	1331	9.5%
Foreign Words	111	0.7%
Permuted Foreign Words	106	0.7%
Total	5525	39.7%

Length of Cracked Passwords		
Length	**Count**	**Percentage**
1 character	9	0.1%
2 characters	7	0.1%
3 characters	91	1.6%
4 characters	307	5.5%
5 characters	415	7.5%
6 characters	1957	35.4%
7 characters	1306	23.6%
8 characters	1433	25.9%
Total		100.0%

As to those passwords which remain unbroken, we can only conclude that these are much more secure and ‘‘safe’’ than those to be found in our dictionaries and permutations. One such class of passwords is punctuated 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 Sparc 2, 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 we don't know what passwords were chosen by those users we were unable to crack, we can say with some surety that it is doubtful that anyone else using this dictionary-based technique could crack them in a reasonable amount of time, either.

3. Action, Reaction, and Proaction

What then, are we to do with these results? 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 [10, 11]. 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 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.* [12] A good password is one that is easily remembered, yet difficult to guess. When confronted with the choice between remembering an easily guessed password and 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 up a 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 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 excellent warders for your system. The drawbacks are that they can be expensive (about $25.00 per user plus an initial setup fee) 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.

4. Overview of 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.

Such a proactive password checker must meet seven criteria:

1) The tests for the password must always be invoked. Otherwise, the tests may be bypassed and a weak password installed on the system. (Most UNIX system password changing programs fail this test, as after three tries weak passwords are allowed [13]).

2) The checker must be able to reject any password in a set of common passwords, or which is a transformation of common passwords. Among the permutations detected in this experiment that such a requirement would eliminate are passwords which:

   - Exactly match a word in a dictionary (not just in the system dictionary)
   - Match a reversed word in a dictionary (with or without capitalization)
   - Match a dictionary word with the letters 'o', 'l', 'z', and 's' replaced by the numbers '0', '1', '2', and '5'
   - Do not contain mixed upper and lower case, or mixed letters and numbers, or mixed letters and punctuation
   - Match a word in a dictionary with some or all of the letters capitalized
   - Match a word in a dictionary with an arbitrary letter turned into a control character
   - Are shorter than a certain length (*i.e.*, all passwords shorter than six characters are disallowed)

   This allows words in a dictionary to be eliminated. This requirement alone would eliminate password cracking if one checked proposed passwords against the dictionary used by attackers. Of course, the problem is acquiring a comprehensive enough dictionary; many large dictionaries are available, but there is no guarantee these have every character sequence that an attacker may try.

3) The checker must allow per-user discrimination in its tests. Among the permutations detected in this experiment that such a requirement would eliminate are passwords based on the user's:

   - Account name
   - Given name or initials

   However, some people have certain associations which may lead to passwords which are easy to guess; for example, the string ''HeidiTu''' is a fairly obvious guess for the first author's password (as his daughter is named ''Heidi Tinúviel'') but the apostrophe makes it an unlikely guess for someone else. This suggests allowing dictionaries to be selected on a per-user basis as well.

4) The checker must allow per-site discrimination in its tests. In some sense, any checker allows this as it can be modified and recompiled. However, the principle of psychological acceptability [14] implies that modifying a set of tests be less cumbersome; so, a configuration file best implements this requirement. This allows the system administrator to turn on certain tests, and modify or disable others (such as the minimum acceptable length for a password).

5) The checker should have a pattern matching facility that can be used in tests. As indicated above, not all bad password choices will be in dictionaries; for example, repetitions of login names typically are not. One could construct a dictionary containing such repetitions, but it is far simpler to describe these by patterns. Such a facility would eliminate passwords which:

- Are based on repetitions of the user's account name
- Consist solely of numeric characters (*i.e.*, Social Security numbers, telephone numbers, house addresses or office numbers)
- Look like a state-issued license plate.
- Are based on repetitions of the user's initials or given name
- Are patterns from the keyboard (*i.e.*, "aaaaaa" or "qwerty")

Note this last example brings in a site dependency (specifically, where the site is located geographically).

6) The checker should be able to run subprograms and use the results in tests. This is particularly useful for eliminating passwords which are:

- Simple conjugations of a dictionary word (*i.e.*, plurals, adding "ing" or "ed" to the end of words, *etc.*)
- Common misspellings of dictionary words (*i.e.*, "stoping" as well as "stopping", "bananna" as well as "banana", *etc.*)
- Made up of two words put together (*i.e.*, "hithere", "goodbye", *etc*)

The subprogram facility has other uses. For example, it can also be used to check for passwords based on local host names.

7) The tests should be easy to set up. If writing a test is a very complex and error-prone procedure, administrators will pick only simple tests which may not help much. As a general principle, security mechanisms should not require much effort to use because if it is not psychologically acceptable the mechanism will either be unused or misused.

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 the research described in section 2 (and others).

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 would 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.

5. Example of A Proactive Checker

The proactive password checker *pwcheck*, a part of the *passwd+* password changing program, provides facilities to meet these requirements. It uses a "little language" to encode tests to determine if a password is too easy to guess. Whenever a password is supplied it runs these tests, and if any test evaluates to the password is rejected and the user told why the password is unacceptable.

5.1. Configuration File

The heart of *pwcheck* is the configuration file, which contains commands to set and evaluate variables and tests to determine if the proposed password is too easy to guess. The tests are composed of expressions, which are in turn made up of constants, variables, and functions. When a user enters a password, it can be stored in a variable. All variables contain strings, and several forms of assignment exist.

When *pwcheck* starts, it automatically sets several variables to values obtained from the user information stored in */etc/passwd*, and from the host:

Predefined User- and Host-Related Variables	
variable	**value**
`user`	user (account) name
`uid`	user identification number
`gid`	(primary) group identification number
`gecos`	user information
`homedir`	home directory
`shell`	user's login shell
`host`	host name (no domain)
`domain`	(internet) domain name, if any
`fqdn`	(internet) fully qualified domain name
`nisdomain`	NIS domain name, if any

Pwcheck also sets several other variables from information gleaned about the password:

Predefined Password-Related Variables	
variable	**value**
`newpwd`	proposed password
`curpwd`	current password (if known)

Values may be assigned to variables using control lines like

```
setvar system "windsor.dartmouth.edu"
```

which assigns to `system` the string "windsor.dartmouth.edu". As a string is a sequence of alphanumeric characters (including underscore), an escaped character, or a quoted string, in this assignment, the quotes are needed because `setvar` assigns the first string following the name of the variable to the variable. Without the quotes, `system` would be assigned the value "windsor". The variable `var` is referenced using the notation `$(var)`; if the variable name is 1 character long, the parentheses can be omitted.

`Setvar` statements do not evaluate the quantity being assigned. To do so, the `evalvar` assignment statement is needed. For example, the function `first(s, t)` takes two strings `s` and `t` as arguments, and returns the numerical position of the first character in `s` that is also in `t`. The function `substr(s, b, e)` returns the substring of `s` beginning at character position `b` and ending at character position `e` (inclusive). So, if `system` contains "windsor", the function

```
substr($(system), 1, first($(system), ".") - 1)
```

evaluates to "windsor"; but saying

```
setvar hostname substr($(system), 1, first($(system), ".") - 1)
```

simply assigns the string "substr($(system), 1, first($(system), ".") - 1)" to `hostname`. The assignment

```
evalvar hostname substr($(system), 1, first($(system), ".") - 1)
```

will evaluate the functions and assign the result "windsor" to the variable.

Finally, one can extract substrings based on pattern matching. Suppose the user information for the user Bishop is stored in the variable `G` as "Matt Bishop,107 Raven House,3267". The control line

```
setpat "$G" "^\\([^,]*\\),\\([^,]*\\),\\(.*\\)$" user off ext
```

assigns "Matt Bishop" to the variable `user`, "107 Raven House" to the variable `office`, and "3267" to the variable `ext`. Note that the second quoted string uses the pattern matching operator "`\(`" and "`\)`" to return that part of the string matched by the pattern between those operators. However, when that string is read, the backslashes would be interpreted as escapes for the parentheses and discarded. So, the backslashes must have an escape character, `\`, put in front of them to prevent them from being discarded. Put another way, the first backslashes are escapes; the second are part of the operators.

A number of functions are available for writing the tests. Rather than describe each one individually, we present and discuss the tests that would detect types of passwords identified as too easy to guess in this study and in [1516,17]., In what follows, the password would be considered easy to guess if the expression evaluates to true (non-zero) and not easy to guess if the expression evaluates to false (zero). Also, we shall assume the variable `user` contains the user's login (account) name, `p` the proposed password, `f`, `m`, `l` the user's first, middle, and last names respectively; `i` his or her initials, via:

```
evalvar i lcase(substr($f,1,1))  \
        lcase(substr($m,1,1)) lcase(substr($l,1,1))
```

and that any dictionaries in use are named *dictionary*.

1) Passwords based on the user's account name.

   Here we check for three of the variations described in section 2.2(1); extensions to other variations are straightforward.

```
"$p" == "^" "$user" "$"
"$p" =~ "^" prot("$user") "[0-9]+$"
"$p" =~ "^." prot("$user") ".$"
```

   Those characters which are not operators are quoted so that the checker will interpret them as part of a string; the variables are quoted because substitution is done before the line is parsed. The operator "==" is the comparison operator, and the operator "=~" matches the string on the left with the pattern on the right. The function `prot(s)` scans the string `s` looking for metacharacters meaningful to the pattern matcher; it returns the string `s` with the appropriate escapes inserted so that `s` is interpreted as a string. (So, for example, if `s` contained "he.l*o", `prot(s)` would return "he\.l*o", as "." and "*" represent "any character" and "0 or more repetitions of the previous character", respectively). The function `lcase(s)` returns the string `s` with all upper case letters made lower case. Placing strings next to one another concatenates them; so if the user's name were "Bishop", these three expressions would be

```
"$p" == "^Bishop$"
"$p" =~ "^Bishop[0-9]+$"
"$p" =~ "^.Bishop.$"
```

The first matches the login name; the second matches any occurrence of the login name followed by 1 or more digits; and the third, the login name surrounded by single characters on either end.†

2) Passwords based on the user's initials or given name.

Again, here we show the tests for the 10 of the variations described in section 2.2(1):

```
"$p" =~ "^\\(" $i "\\)*$"
"$p" == substr($f,1,1) $l
"$p" == fcase(substr($f,1,1)) fcase($l)
"$p" == rev(fcase($f))
"$p" == rev(fcase($l))
"$p" == fcase(substr($f,1,1)) fcase(substr($m,1,1)) fcase($l)
"$p" == fcase($f) fcase(substr($l,1,1))
"$p" == fcase($i) rev(fcase($i))
"$p" == ucase($f) "-" ucase($l)
"$p" == "$l" ucase(substr($f,1,1))
```

Suppose the user's given name is "Matthew A. Bishop"; then `f` contains "Matthew", `m` contains "A.", `l` contains "Bishop", and `i` contains "mab". The first line returns 1 if the password is 0 or more repetitions of the initials, using the pattern-match operator "=~". The second line returns 1 if the password is "MBishop"; notice the operator is now "==", which tests for equality. The third line returns 1 if the password is "mbishop"; the fourth, if the password is "wehttam" (the function `rev(s)` reverses the string `s`); the fifth, if the password is "pohsib"; the sixth, if the password is "mabishop"; the seventh, if the password is "matthewb"; the eighth, if the password is "MabbaM"; the ninth, if the password is "MATTHEW-BISHOP"; and the tenth, if the password is "BishopM". Obviously many more permutations are possible.

3) Passwords which exactly match a word in a dictionary (not just system ones).

If the dictionary is an unsorted file with one word per line, the expression

```
infile("$p", dictionary)
```

returns 1 if the value of the variable p is one of the lines of the file. If the dictionary's lines are sorted in ascending ASCII order, the binary search function

```
inbinfile("$p", sort_dictionary)
```

is considerably faster. Finally, the database may be stored in a format enabling very rapid searches; a function is provided to take advantage of this. Note that each of these functions search the file directly rather than by using a subcommand, both for speed and to avoid making the proposed password visible to other processes.

4) Passwords which match a reversed word in the dictionary.

This is the same as asking if the reversed password is in the dictionary:

```
infile(rev("$p"), dictionary)
```

5) Passwords which match a word in the dictionary with some or all letters capitalized.

Here, we just treat all characters as lower-case. If the password, with all letters lower case, appears in a version of the dictionary with all letters lower case, we want the expression to evaluate to 1. The simplest way to do this is to use the subcommand execution facility:

```
inprog(lcase("$p"), "tr A-Z a-z < dictionary")
```

The *tr*(1) command is executed and each line of output is compared to the lower case password. If any are equal, the expression evaluates to 1. (As an efficiency measure, storing the dictionary words in lower case eliminates the need for using *tr*.)

6) Passwords which match a reversed word in the dictionary with some or all letters capitalized.

This is just like the previous expression, but the password is reversed:

```
inprog(rev(lcase("$p")), "tr A-Z a-z < dictionary")
```

7) Passwords which match a word in a dictionary with an arbitrary letter turned into a control character.

Here, we simply change all control characters in the password to their letter equivalent. (We could implement this expression exactly by looking for the first control character and using that, then the second, and so on, but that is much more complicated as the little language has no iteration function.) We then compare the results to the dictionary, as before:

```
infile(trans("$p", controls, "A-Z[\\]^_"), dictionary)
```

In the little language, the distinguished constant `controls` is a string of all control characters except ASCII NUL (which is used as a string terminator).

8) Passwords which match a dictionary word with the following translations, either alone or in various combinations: ‘l’→‘1’, ‘o’→‘0’,‘s’→‘5’, ‘z’→‘2’

   Here we simply give some examples, as there are 15 transformations possible:

```
infile(trans("$p", "0125", "olzs"), dictionary)
infile(trans("$p", "02", "oz"), dictionary)
infile(trans("$p", "012", "olz"), dictionary)
infile(trans("$p", "15", "ls"), dictionary)
```

9) Passwords which are simple conjugations of a dictionary word (*i.e.*, plurals, adding “ing” or “ed” to the end of the word, etc.).

   This type of password is really just a part of speech; the simplest way to look for it is to use the spelling checker. If the word is incorrectly spelled, it will be printed to the output of *spell*(1); otherwise, nothing is printed and no check is performed:

```
!inprog("$p", "spell -h /dev/null", "$p")
```

   This says to run the program *spell*(1), giving it as input the password (the second `$p`). If the password (the first `$p`) is in the output, the expression evaluates to 0 (the “!” negates the value of the function). At no time is the input placed on a command line, so the above test would not reveal the password even to a process status list.

10) Passwords which are patterns from the keyboard (*i.e.*, “aaaaaa” or “qwerty”).

   This can best be done by building a dictionary of such sequences. Note that a dictionary can contain patterns; for example, to eliminate all sequences of repeated characters, place a line containing the pattern “`^\(.\)\(\1\)*`” in the dictionary, and use the function `filepat`:

```
filepat("$p", patternfile)
```

This returns true 1 if the password matches any pattern in the file `patternfile` (which has one pattern per line). Note only one backslash is needed in the pattern because when the file containing the pattern is read, each line is treated as a complete pattern; it is not broken into strings.

11) Passwords which are shorter than a specific length (*i.e.*, nothing shorter than six characters).

The function `length` returns the length of a string:

```
length("$p") < 6
```

evaluates to 1 when the password is shorter than 6 characters.

12) Passwords which consist solely of numeric characters (*i.e.*, Social Security numbers, telephone numbers, house addresses or office numbers).

A pattern can best describe this type of password:

```
"$p" =~ "^[0-9]+$"
```

13) Passwords which do not contain mixed upper and lower case, or mixed letters and numbers, or mixed letters and punctuation.

Expressions to look for these use the arithmetic and logical operators in tests:

```
!ismixed("$p") || nnotalphas("$p") > 0
```

The operator "||" is the logical "or" operator. This expression has a value of 1 for all passwords without mixed case (`!ismixed`), or which do not have some non-alphabetic character (`nnotalphas`). A better form of this expression would evaluate to 1 for any password which does not contain at least one alphanumeric:

```
nnotalnums("$p") > 0
```

14) Passwords which look like a state-issued license plate.

The formats of license plate numbers vary from state to state (a good example of why per-site discrimination is needed). In New Hampshire, license plates for cars are either 4, 5, or 6 digits, or three letters followed by three digits:

```
"$p" =~ "^[0-9]{4,6}$" || "$p" =~ "^[A-Za-z]{3}[0-9]{3}$"
```

In Pennsylvania, autombile license plates are three letters followed by 3-4 digits:

```
"$p" =~ "^[A-Za-z]{3}[0-9]{3,4}$"
```

15) Passwords made up of 2 words.

The function `mwords` returns 1 if its first argument can be split into two strings both of which are in the dictionary named in its second argument. For example, the expression `mwords("hithere","/usr/dict/words")` returns 1 as “hi” and “there” are both in the file */usr/dict/words*. The expression to use is simply:

```
mwords("$p", "/usr/dict/words")
```

16) Passwords which are different than previous passwords

If passwords were stored in cleartext, the risk of compromise would be tremendous. So, a proposed password can be compared to a set of previous passwords stored in hashed form:

```
filecrypt("newpassword","file_of_hashes")
```

returns 1 if “newpassword” is a password in the file “file_of_hashes”. Passwords can be stored in hashed form using the function `crypt`, as in

```
inprog("", "/bin/cat >> file_of_hashes", crypt("passwword","random"))
```

which executes a command to append to “file_of_hashes” (argument 2) the result of hashing the password “password”, and returns 1 if there is no output from the appending (which there should not be).

17) Passwords which have too many characters in common with their immediate predecessor

This criterion can involve the characters, or the characters and position. For example, if the current password is “hello” and the proposed one is “holl0-r”, the function will return 4 (because the two arguments have 4 characters in common, ignoring position), and the function returns 3 (because the “h” and the two “l”s match in position, but the “o” does not have the same position in both arguments).

5.2. Tests and Associated Controls

Expressions are used in tests to determine if a password is too easy to guess. Associated with the tests are statements to be printed if the test succeeds (to inform the user why the password is being rejected), if it fails (to inform the user of the criteria passed), if the user asks for help (for educational purposes), and error handlers (as the test may use an unavailable resource, such as a dictionary not present on the system)..

As an example, consider the requirement that all passwords be at least 7 characters long and not be in the system dictionary:

```
# test length
test
eval length("$p") < 7
iftrue "Your password is too short"
iffalse "Your password is long enough"
help "Your password must be at least 7 characters long"
endtest
# test for in the dictionary
test
eval infile("$p", "/usr/words/dict")
onerror true
iferror "Could not access /usr/dict/words -- try again later"
iftrue "Your password is in the dictionary"
iffalse "Your password is not in the dictionary"
help "Your password must not be in the dictionary (use non-alphanumerics)"
endtest
```

The first line in the first test block says that the expression is to be evaluated and if it evaluates to true (nonzero), the password is to be rejected. If the password stored in the variable `p` is ‘‘aardvark’’, the expression will evaluate to true. If the test is true, the message on the next line beginning with *iftrue* is printed; if false, the message on the next line beginning with *iffalse* is printed. In this case, the message ‘‘Your password is long enough’’ will be printed. Had the password been ‘‘hello’’, the test expression would evaluate as false, and the alternate message ‘‘Your password is too short’’ would be printed. The next line, *help*, contains a string to be printed when the program is run in help mode.

The next block shows how to check for words in a dictionary. The expression in the *eval* line is evaluated; the password ‘‘aardvark’’ would be found in the dictionary, rejected, and the message ‘‘Your password is in the dictionary’’ would be printed. If an error occurs (because ‘‘/usr/dict/words’’ is not available, for instance), the message ‘‘Could not access /usr/dict/words -- try again later’’ will be printed. The line containing ‘‘`onerror true`’’ says to treat an error condition as though the test evaluated true, (and so the proposed password would be rejected). In an error condition, however, the ‘‘iftrue’’ message would not be printed.

Consider instead the password "I_lxp:r". It (most likely) is not in the dictionary because it contains characters other than a letter or digit. Doing the lookup can take quite a bit of time, though. Because the expression language uses lazy evaluation of "&&" and "||", the test could be rewritten as

```
eval nalnum("$p") == length("$p") && infile("$p", "/usr/dict/words")
```

If the first part were false, (*i.e.* the password contains non-alphanumeric characters), then the second (the dictionary lookup) will not be evaluated.

The `iftrue`, `iffalse`, `iferror`, and `onerror` controls apply to the test block in which they appear only (system defaults are provided if they are absent). The default block overrides these, and remain in force until changed by another such block:

```
# default tests and actions
default
onerror true
iftrue "The password is too easy to guess"
iferror "An error occurred; contact the system administrator"
help "Use memo #234 to guide you in selection of your password"
enddefault
```

Finally, if the test block contains only the test (an `eval` line), the block can be collapsed into a single line by putting the test on the same line as `test`. So,

```
test
eval length("$p") < 2
endtest
```

and

```
test length("$p") < 2
```

do exactly the same thing.

5.3. Miscellaneous Controls

Several miscellaneous controls tailor the expression evaluation and configuration files as desired. The pattern matcher used above is the GNU pattern matcher; if one were more familiar with the Berkeley pattern matcher (which is the same as the Version 7 pattern matcher), one could use that by having a line of the form

```
pattern bsd4
```

at the top of the configuration file.

Secondly, UNIX passwords are truncated at eight characters; so if the password is ‘‘ambiguously’’, this could be guessed (since ‘‘ambiguous’’ is in the system dictionary, and the two words have the same first 8 letters. So, the control line

```
complen 8
```

forces all string comparisons to stop after the first 8 characters. Note this does *not* affect pattern matching, because the length of the pattern may depend upon the string being matched (for example, if the partial string match operators are used).

5.4. Summary

The proactive password checker *pwcheck* offers facilities of enough power to detect those passwords which are likely to be guessed easily. As with any measure that seeks to counter a threat, the changing nature of the dictionaries used to guess passwords means that no proactive checker can prevent all passwords from being guessed; however, experience with the predecessor of *pwcheck* has shown the use of such a checker, combined with sufficiently powerful rules, does lessen the success of attackers compromising passwords.

6. 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 break-in problem occurs.

References

[1] Robert T. Morris and Ken Thompson, ‘‘Password Security: A Case History,’’ *Communications of the ACM*, vol. 22, no. 11, pp. 594-597, November 1979.

[2] ‘‘Proposed Federal Information Processing Data Encryption Standard,’’ *Federal Register (40FR12134)*, March 17, 1975.

[3] Matt Bishop, ‘‘An Application of a Fast Data Encryption Standard Implementation,’’ *Computing Systems*, vol. 1, no. 3, pp. 221-254, Summer 1988.

[4] David C. Feldmeier and Philip R. Karn, ‘‘UNIX Password Security – Ten Years Later,’’ *CRYPTO Proceedings*, Summer 1989.

[5] Philip Leong and Chris Tham, ‘‘UNIX Password Encryption Considered Insecure,’’ *USENIX Winter Conference Proceedings*, January 1991.

[6] Daniel V. Klein, ‘‘‘‘Foiling the Cracker’’ – A Survey of and Improvements to UNIX Password Security,’’ *Proceedings of the USENIX Security Workshop*, Summer 1990.

[7] Eugene H. Spafford, ‘‘The Internet Worm Program: An Analysis,’’ Purdue Technical Report CSD-TR-823, Purdue University, November 29, 1988.

[8] Bruce L. Riddle, Murray S. Miron, and Judith A. Semo, ‘‘Passwords in Use in a University Timesharing Environment,’’ *Computers & Security*, vol. 8, no. 7, pp. 569-579, November 1989.

[9] Ana Marie De Alvare and E. Eugene Schultz, Jr., ‘‘A Framework for Password Selection,’’ *USENIX UNIX Security Workshop Proceedings*, August 1988.

[10] T. Raleigh and R. Underwood, ‘‘CRACK: A Distributed Password Advisor,’’ *USENIX UNIX Security Workshop Proceedings*, August 1988.

[11] Alec Muffett, *Crack*, 1992. Available via anonymous *ftp* from *cert.org*.

[12] Dr. Brian K Reid, DEC Western Research Laboratory, 1989. Personal communication.

[13] *UNIX User’s Reference Manual, 4.3 Berkeley Software Distribution -11 Version,* Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, April 1986.

[14] Jerome Saltzer and Michael Schroeder, ‘‘The Protection of Information in Computer Systems,’’ *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, September 1975.

[15] Harold Joseph Highland, ‘‘Random Bits and Bytes: Testing a Password System,’’ *Computers and Security*, vol. 11, no. 2, pp. 110-113, April 1992.

[16] Alec Muffett, *Crack*, 1992. Available for anonymous *ftp* from *cert.org*.

[17] Aeleen Frisch, *Essential System Administration,* O’Reilly and Associates, Sebastopol, CA, 1991.