OPUS: Preventing Weak Password Choices*

Eugene H. Spafford†

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA

A common problem with systems that use passwords for authentication results when users choose weak passwords. Weak passwords are passwords that are easy to guess, or likely to be found in a dictionary attack. Thus, the choice of weak passwords may lead to a compromised system. Methods exist to prevent users from selecting and using weak passwords. One common method is to compare user choices against a list of unacceptable words. The problem with this approach is the amount of space required to store even a modest-sized dictionary of prohibited password choices. This paper describes a space-efficient method of storing a dictionary of words that are not allowed as password choices. Lookups in the dictionary are O(1) (constant time) no matter how many words are in the dictionary. The mechanism described has other interesting features, a few of which are describe here.

Keywords: Passwords, Dictionaries, Password aging.

1. Introduction

Passwords are a commonly used method of authentication. A unique sequence of characters is presented to the system when identification is needed. This sequence is then compared with a stored sequence, perhaps after some transformation (e.g., encryption). A match provides the proof of identity.

One weakness with password systems is the choice of the password. If the choice of possible characters to use in the password is too small, or if the overall length of the password is too short, the password may be compromisable. Even a rich character set may not be sufficient to create secure passwords if the combination of characters is restricted to an arbitrary set of possibilities. Thus, good password choice should avoid common words and names [1, 6, 10, 12, 15].

As an example, consider the UNIX[1] password system [12]. The current password mechanism is based on a cryptographic transformation of a fixed string of zero bits, using the user-supplied password as a key. The transformation is an altered version of DES encryption, performed 25 times. The transformation is sufficiently slow so that exhaustive keyspace attacks are currently not practical, although fast implementation such as *deszip* [3] can perform many thousands or tens of thousands of comparisons per second.

In UNIX, the encrypted version of the password has traditionally been kept in a world-readable file; the safety of the passwords has been protected by the time complexity of an exhaustive attack. Thus, one of the keys to the safety of UNIX passwords is a large potential keyspace for passwords. If the full character set is used, and seven or eight-character

*This paper will appear as [22].

†E-mail: spaf@cs.purdue.edu.

[1] UNIX is a trademark of Unix System Laboratories, Inc.

passwords are chosen, the number of potential passwords to be searched is far too large to be successfully searched, even at high speed.[2] Unfortunately, users often select passwords that do not exploit the large keyspace available. Instead, they choose common words and names, or simple transformations of those names. This greatly simplifies an attacker's task.

This tendency to select weak[3] passwords has led to a number of system break-ins, some quite highly publicized [14, 18, 20, 21, 24]. Current technology is such that construction of a large pre-encrypted dictionary on-line using optical disks is easily done. By creating such a dictionary, a password search and attack may be easily conducted in a matter of seconds. Without such a database, but using a tool such as *deszip* on a modern workstation, it is possible to make a full scan of 300 000 dictionary entries against several hundred passwords in a matter of a few hours or days.

Despite widespread publication of good password policy and the risks inherent in bad passwords, users continue to select weak passwords. This is a continuing threat to the best-managed systems [2, 7–11, 15, 19, 23, 25].

There are four basic methods for a system administrator to enforce better password security on a computer system:

(1) Educate and encourage users to make better choices of passwords.

(2) Generate strong passwords for users and do not allow them to choose passwords of their own creation. This is often done using some random password generator.

(3) Check passwords after-the-fact and force users to change those that can be easily broken with a dictionary attack.

(4) Screen users' password choices and prevent weak ones from being installed.

This first method, that of educating users to choose strong passwords, is not likely to be of use in environments where there is a significant number of novices, or where turnover is high. Users might not understand the importance of choosing strong passwords, and novice users are not the best judges of what is "obvious". For instance, novice users (mistakenly) may believe that reversing a word, or capitalizing the last letter, makes a password "strong".

A further problem is if the education provided to users on how to select a password is itself dangerous. For instance, if the education provided gives users a specific way to create passwords—such as using the first letters of a favorite phrase—then many of the users may use that exact algorithm, thus making an attack easier.

The second method of strengthening passwords is to generate the passwords for the users and not allow them the opportunity to select a weak password. For this mechanism to work well the passwords need to be randomly drawn from the whole keyspace. Unfortunately, this method also has flaws. In particular, the "random" mechanism chosen might not be truly random, and could be analyzed by an attacker. Furthermore, random passwords are often difficult to memorize (especially if they are changed (*aged*) regularly). As a result, users may write the passwords down, thus providing an opportunity to intercept them without the effort of a dictionary search.

The third method of preventing poor password choice is to scan the passwords selected, after they are chosen, to see if any are weak. This is supported by many systems, including *deszip* and Cops [5]. There are significant problems with this approach:

[2] Assuming a usable character set of 120 characters, there are 43 359 498 756 302 520 (4.34e16) possible passwords of length one through eight. At 50 000 attempts per second, an exhaustive search of this keyspace would require over 27 480 years to complete.

[3] Strength being defined as the ability to resist a dictionary-based attack, and weakness as its opposite.

- The dictionary used in the search may not be comprehensive enough to catch some weak passwords. Outside attackers might think of these choices, but the password scanner would not include them in the search.

- The scanning approach takes time, even for a fast implementation. A lucky (or determined) attacker may be able to penetrate a system through a weak password before it is discovered by the scanner. This is especially a problem in an environment with a very large number of users.

- The output of a scanner may be intercepted and used against the system.

Additionally, there is not always a correlation between finding a weak password and getting it replaced with a stronger one. At many universities, for example, faculty members have repeatedly been informed of the weakness of their passwords as exposed by a scanner, but they have not chosen new passwords in years. The administration of university systems is such that it is impossible to force faculty members to choose better passwords.

The fourth method, that of disallowing the choice of poor passwords in the first place, appears to have none of the drawbacks mentioned above. However, it too has difficulties associated with it. In particular, the storage required to keep a sufficiently large dictionary may prevent this method from being used on workstations and small computer systems. For instance, the standard UNIX dictionary, */usr/dict/words*, is about 25 000 words and 200 000 bytes of space. A dictionary of 10–20 times that size would be necessary for reasonable protection; there are over 170 000 words in Webster's New World Dictionary, and that would occupy well over a million bytes of disk storage. That figure does not include many slang and colloquial words and phrases, nor does it include any user names, local names and phrases, likely words in foreign languages, or other strings shown to be poor password choices. A moderately comprehensive dictionary I have used in password research has over 500 000 entries, and requires almost five million bytes of storage.

Maintaining such a large dictionary is also difficult. To add new words or phrases means that the dictionary must have additional space overhead for indexing or it must be sorted after each addition—otherwise, lookups take time proportional to the length of the dictionary. In small computer environments, neither of these alternatives may be appropriate.

2. OPUS

The OPUS Project[4] is intended to address the space problems associated with a sufficiently complex password screening dictionary. The goal is to derive a mechanism that provides protection equivalent to a comparison against a large dictionary, yet be small enough to be practical in a small computer environment.

2.1. The Dictionary Filter

The central component of this system is a Bloom filter-encoded version of the dictionary [4]. A Bloom filter is a well-studied probabilistic membership checker, often used in applications such as spelling checkers [13, 16, 17]. It works as follows: a word to be entered into the filter is passed through *n* independent hash functions generating integer values. Each of these values is used as an index into the filter, represented as a bitmap. The bits (one per hash function) corresponding to the input word are then set. This procedure is repeated for each word to be entered into the filter.

When a lookup is to be performed, the word to be examined is passed through the same hash functions and the corresponding bits in the filter are examined. If any of the bits is reset (i.e., not set), then the word is determined not to be present in the dictionary. If all the corresponding bits are set, the likelihood is high that the word was in the list

[4] Obvious Password Utility System.

that was used to build the dictionary. In the case of Opus, this means the choice is rejected as a weak password choice. The probability of a false rejection can be set arbitrarily low by increasing the size of the bitmap and increasing the number of hash functions used; an obvious upper bound on the size of the hash table is the size of the plaintext dictionary.

To be more exact, assume we have a hash table of N bits, and d independent hash functions. From [4], with n words we have the proportion of bits left unset, ϕ, equal to

$$\phi = \left(1 - \frac{d}{N}\right)^n$$

A word will be falsely shown as present in the dictionary if and only if it hashes to a set of bits that are all set. The expected proportion, P, of words in the input space that will be mistakenly shown as in the dictionary is thus

$$P = (1 - \phi)^d$$

From these equations, we can derive appropriate values to choose for our filter and hash functions.

For example, suppose we pick $n = 250\,000$ words for the dictionary, and we wish to have a 0.5% chance ($P = 0.005$, i.e. one out of 200) of false positives on any given text string. If we choose six uniform hash functions, we will need 2 800 000 bits of storage and achieve $\phi = 0.586$. This works out to a file of 350K bytes. Doubling the chance of false positives to 1% ($P = 0.01$) results in needing only 300K bytes of storage for the dictionary with six hash functions. Storing the full dictionary as plaintext would likely take in excess of 2 Mb of storage. Thus, we are able to achieve almost a sevenfold compression with only a small loss of accuracy.

As can be seen from the above examples, with the appropriate choice of hash functions it is possible to greatly reduce the storage necessary to keep an extensive dictionary of words to compare against password choices. By making queries on the dictionary with variations of the candidate password—upper/lower case, reversed, trailing digit, etc.—it should be possible to quickly check for the strength of the password. Each probe into the dictionary is basically a constant-time operation, so the number of words in the dictionary has no effect on the time of access. If the union of all the probes results in a positive response, the user is told to try again.

2.2. Other Features

The model of the dictionary used in Opus provides benefits other than simple dictionary lookup. By providing a writable interface to the dictionary for the system administrator, it is a simple task to add the representation of new words to the dictionary. The administrator can therefore augment the dictionary with local user names and colloquialisms. Adding words to the dictionary requires no expensive sorting or temporary storage. Furthermore, the system administrator never needs to be concerned if a word has already been added—adding a word more than once has no effect.

The Opus system also supports password aging. With password aging, users are required to change their passwords periodically. However, a common fault with password aging is that users attempt to reuse old passwords, and this may present a security risk.

Opus can be configured so that whenever a password is changed, it is added to the dictionary. Thus, if a user attempts to reuse an old password, he or she will find it already in the dictionary, and the choice will not be allowed. As seen from the value of ϕ, above, there is plenty of room in the dictionary for adding new words, so even prolonged operation will not result in a noticeable degradation of service. Also, simple steps need to be taken to prevent very frequent changes of passwords that might degrade the filter, such as putting a minimum time for which a new password must be kept before a change is again allowed.

One obvious problem with updating the dictionary in this manner is the possibility of an attacker using delta information to craft a set of password attempts. That is, by observing the changes made to the filter when another user changes his password, an attacker might be able to use the hash functions to derive a set of possible text strings that account for the changes, and use these in a penetration attempt.

A related problem is if an attacker finds a way to use the dictionary as a filtering mechanism to exclude patterns when doing a brute-force keyspace search to break passwords. Doing a probe into the dictionary will determine if a candidate is a possible choice or not, thus saving (some) on the computation required to perform an exhaustive search.

Luckily, there is a simple way to defeat these problems. Instead of hashing plaintext words into the dictionary, Opus first encrypts the words to be entered or examined. The encryption must be something time consuming, similar to multiple rounds of the DES function, and computationally infeasible to reverse. The hashing algorithms are then applied to the encrypted string rather than to the plaintext. Thus, to gain any information from the dictionary, either as a pre-screen or as a source of delta information, would require much more computational effort than some other approach (e.g., exhaustive keyspace search).

To further confound attackers, the key used to encrypt the input words should either be site selectable, or generated as a function of the input word itself. For instance, if something similar to the Unix mechanism is used, the first and last letter of the input word, converted to upper case, could be used as the "salt." As there is never a reason to recover words from the dictionary, this choice of key is something that probably cannot be recovered unless the plaintext word is known.

3. Final Remarks

This paper has discussed the motivations and design behind a system for preventing users from installing weak passwords. The system should be compact and simple to customize and enhance. It can be used standalone, as a front-end to an existing password program, or coupled with some form of password generator so as to prevent the accidental generation of a word susceptible to dictionary attacks.

The choice of hashing algorithms used with the system is critical for the success of the filter. Choosing non-uniform or overlapping hash algorithms reduces the effectiveness of the Bloom filter by increasing the incidence of false positives (effectively shrinking the number of useful bits employed). When possible, the hash algorithms should be chosen to produce the same results whether used on a string or on its reverse. This will allow probes for common words and their reverses to be made simultaneously. Case insensitivity can also be used in the hash functions, but this may result in too great a narrowing of the keyspace; words in monocase, or with only a leading or trailing capital letter, are perhaps the only combinations that need to be examined.

A Unix version of Opus is being constructed. It will be preloaded with a locally developed dictionary of almost 500 000 strings. Experiments will then be conducted to determine, for this dictionary, the optimal working size and number of hash functions. Further experiments will determine the accuracy rate for rejection of candidate passwords that are not present in the real dictionary, and the speed of operation. By performing side-by-side experiments with users selecting potential passwords and comparing a dictionary search with the results of the Bloom filter, it should be possible to determine the overall utility of this approach.

References

[1] A. M. De Alvaré, *How crackers crack passwords, or what passwords to avoid, Tech. Rep. UCID-21515*, Lawrence Livermore National Laboratory, 1988.

[2] A. M. De Alvaré and E. E. Schulz Jr., *A framework for password selection, Tech. Rep. UCRL-99382*, Lawrence Livermore National Laboratory, 1988.

[3] M. Bishop, An application of a fast data encryption standard implementation, *Comput. Syst., 1* (1988) 221–254.

[4] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13 (July 1970) 422–426.

[5] D. Farmer and E. H. Spafford, The COPS security checker system. In *Proc. Summer Usenix Conf.*, Usenix Association, June 1990.

[6] S. Garfinkel and E. H. Spafford, *Practical Unix Security*, O'Reilly and Associates, Inc., Sebastapol, CA, May 1991.

[7] D. L. Jobusch and A. E. Oldehoeft, A survey of password mechanisms: weaknesses and potential improvements, part 2, *Comput. Secur.*, 8 (1989) 675–689.

[8] D. L. Jobusch and A. E. Oldehoeft, A survey of password mechanisms: weaknesses and potential improvements, part 1, *Comput. Secur.*, 8 (1989) 587–603.

[9] D. V. Klein, A survey of, and improvements to, password security. In *UNIX Security Workshop II*, 5–14, Usenix Association, August 1990.

[10] B. Menkus, Understanding password compromise, *Comput. Secur.*, 7 (December 1988) 549–552.

[11] J. M. Carroll et al., The password predictor: a training aid for raising security awareness. *Comput. Secur.*, 7 (October 1988) 475–481.

[12] R. Morris and K. Thompson, Password security: a case history. In *Unix Programmer's Supplementary Documentation*. AT&T, Murray Hill, NJ, November 1979.

[13] J. K. Mullin, A second look at Bloom filters, *Commun. ACM*, 26 (August 1983) 570–571.

[14] N. Munro, Simple password opens navy computer to hacker, *Government Comput. News*, 7 (July 1988) 61.

[15] National Computer Security Center, Password management guideline, *Tech. Rep. CSC-STD-002-85*, U.S. Department of Defense, 1985.

[16] R. Nix, Experience with a space efficient way to store a dictionary, *Commun. ACM*, 24 (May 1981) 297–298.

[17] M. V. Ramakrishna, Practical performance of Bloom filters and parallel free-text searching, *Commun. ACM*, 32 (October 1989) 1237–1239.

[18] B. Reid, Reflections on some recent computer break-ins. *Commun. ACM*, 30 (February 1987) 103–105.

[19] B. L. Riddle, M. S. Miron and J. A. Semo, Passwords in use in a university timesharing environment, *Comput. Secur.*, 8 (1989) 569–578.

[20] D. Seeley, Password cracking: a game of wits. *Commun. ACM*, 32 (June 1989) 700–703.

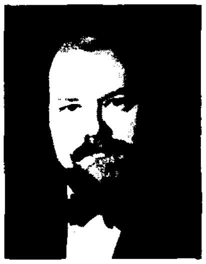
[21] E. H. Spafford, The Internet worm: crisis and aftermath, *Commun. ACM*, 32 (June 1986) 678–687.

[22] E. H. Spafford, Preventing weak pasword choices. In *Proc. 14th National Computer Security Conf.*, NIST/NCSC, October 1991.

[23] C. Stoll, How secure are computers in the U.S.A.? An analysis of a series of attacks on MilNet computers. *Comput. Secur.*, 7 (1988) 543–547.

[24] C. Stoll, *The Cuckoo's Egg*, Doubleday, New York, October 1989.

[25] P. H. Wood and S. G. Kochan, *Unix System Security*, Hayden Book Company, Carmel, IN, 1987.



Gene Spafford is an assistant professor with the Department of Computer Sciences at Purdue University, and is affiliated with the Software Engineering Research Center (SERC) there. At Purdue and SERC, he does research on issues relating to increasing the reliability of computer systems, and the consequences of computer failures. This includes work with computer security, software testing and debugging, and issues of liability and professional ethics.

Professor Spafford is on the editorial or advisory boards of the publications *Computing Systems, Virus Bulletin, Journal of Information Systems Security* and the *International Journal of Computer and Software Engineering*. He is co-author of the books *Practical Unix Security*, O'Reilly and Associates (1991), and of *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats*, ADAPSO (1989).

He is a member of the Association for Computing Machinery (ACM) and is chair of the ACM Self-Assessment Committee. He is a Senior Member of the IEEE and Computer Society of the IEEE, a member of the Usenix Association, and an associate of the National Research Center on Computers and Society.