A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack

Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

November 1994

ABSTRACT

We present a simple scheme that makes guessing passwords based on one-way functions 100 to 1000 times harder. The scheme is easy to program and easy to incrementally add to existing schemes. In particular, there is no need to switch to it all at the same time. Old passwords will still work and have the same security as before (one will not be able to distinguish them from new passwords); newly-entered passwords will become much more secure. The new scheme is independent of the one-way function used and does not require changing any part of the encryption mechanism.

1. Introduction

Password schemes that are based on one-way functions work as follows. When a password is entered, the one-way function is applied to it and the result is stored in a password file. The original password is discarded by the system and no record of it is kept. One-way functions are functions that can be computed easily, but cannot be inverted easily. An encryption scheme is an example of a one-way function (although encryption schemes must also provide a way to decrypt messages through a secret decryption key). To verify a given password, the same one-way function is applied and the result is checked against the stored result. The password file itself need not be kept secret, because one cannot decipher the original passwords from what is stored in the file. At least, that was the idea when this scheme was developed and later incorporated in the UNIX system (Morris and Thompson [MT79] list Wilkes [Wi68] for the origin of this scheme). The password file itself (/etc/passwd) was made available for anyone (who has access to the system) to read.

[1] Supported in part by NSF grant CCR-9301129, and by the Advanced Research Projects Agency under contract number DABT63-93-C-0052.

The main weakness of such a scheme is that users tend to select easy to remember passwords. While it may not be possible for someone to invert the one-way function, it is possible to make lots of guesses and to verify them. A typical attack consists of copying the password file and trying a long list of words (e.g., from the dictionary) until a match occurs. Since this guessing game can be done ‘‘off-line,’’ there is no way to limit the number of guessing attempts. Such attacks were started as soon as UNIX became popular, and have become a major threat to the security of UNIX. Many if not most attacks on UNIX involved grabbing the /etc/passwd file at some point (Cliff Stoll’s account [St89] is particularly enjoyable). Many studies have shown that users tend to select easy-to-guess passwords [JO89]; some found as many as 85% of the passwords on a system to be guessable in reasonable time.

The most common remedy up to date has been for system administrators to check passwords either when they are originally entered and/or on a regular basis using large dictionaries. In the former case, certain rules are imposed that make the password unlikely to appear in any list of guesses; for example, the password is compared to the dictionary, or the system insists that at least on non-alphanumeric character appears somewhere in the middle of the password. In my department, one gets the following message, which is typical, when one tries to use a weak password:

Password must contain at least two alphabetic characters and at least one numeric or special character.

In the latter case, system administrators try themselves to crack passwords on a regular basis, and inform users whose passwords are broken to change it.

This state of affairs has generated a race to collect bigger and bigger dictionaries, and to update them all the time with names in the news, phrases, new categories, etc. ([Sp92]). Accounts of this race has even reached the popular press [Ha91]. It’s not clear who is winning this race, but it is clear that it is a tough one. System administrators who are not up-to-date lose to hackers who are. Making sure that no password appears in the dictionary may not be sufficient, as guesses can automatically generate strings that sound good, or strings that are within one character of dictionary words [MW94]. What makes it so difficult is the advances in DES encryption, which allows for hundreds of guesses a second! When the password scheme for UNIX was first developed, an old encryption scheme was used, but was deemed too risky because encryption took about 1 ms on a PDP11/70. So instead, a scheme based on the DES standard was used. As pointed out by Feldmeier and Karn [FK89], current fast implementations of DES and current fast machines allow encryption to be even faster today than the one rejected 15 years ago.

In this short paper, we suggest a simple scheme that should tilt the playing field back to the ‘‘good guys’’ favor. In will make guessing much more difficult. The scheme is easy to incorporate, it does not involve major changes, it is independent of the one-way function used, and it provides much greater security. The rest of the discussion centered around UNIX, which is the most widely used operating system to rely on one-way functions for password security. The same principles would apply to any such system.

2. Password Security in UNIX

We give a very brief outline of the password mechanism in UNIX. More details can be found in many places (e.g., [MT79], [Kl90], [GS91], [Cu92]).

A UNIX password is actually not encrypted, but used as the encryption key to the *crypt* function. Crypt is using 25 rounds of modified DES[2] to encrypt a string of 0's with the password as a key. The result, with one extra thing to be mentioned shortly, is stored in the /etc/passwd file. Although the strength of one round of DES has been questioned lately, 25 rounds still seems reasonably secure and it is very widely used. There is no published report of anyone breaking crypt to obtain passwords, compared to hundreds of reports of compromised passwords due to guessing.

The possibility of wholesale guessing was recognized early on. Of particular worry was the risk of someone collecting a large list of encrypted likely passwords and circulating the list. This will make it extremely easy for anyone to guess, because no encryption is needed, only comparison of the /etc/passwd file against the list. A list with a million passwords will require less than 20Mbytes to store, which is not a major problem. So an extra protection was added to UNIX passwords. A randomly-generated 12 bit number, called the *salt*, was incorporated into the password scheme. The salt is used to permute some bits during the encryption process, such that the encrypted result depends on the password and the value of the salt. The salt is stored in the open along with the encrypted password in the /etc/passwd file. To verify a given password for a given account, the /etc/passwd file is consulted, the salt for that account is used in the encryption process, and the final result is compared to what is stored in /etc/passwd. This does not make it harder to guess one password, but it makes it 4096 times harder to build a list of encrypted passwords (because every password on the list will have to be used with every possible salt). Suppose that the /etc/passwd contains 500 passwords. Without the salt, each guess can be encrypted only once and compared to 500 potential matches. With the salt, there will be close to 500 different salts (slightly less because of the small chance of duplicates), and each guess will have to be attempted with all these salts separately.

In recent years, more and more sites have used what is called a *shadow password file*, in which the encrypted passwords are read protected. This gives added security, especially against non-sophisticated intruders, but it is not a perfect solution. For one thing, since the passwords need to be read into memory when they are compared, and it is easy to trigger such a comparison, it is not easy to totally protect them. Another problem is the need for other programs, besides the login procedure, to authenticate users. Shadow password files will either cause less authentication, which diverts the weaknesses of the system to the other programs, or will cause more distribution and thus less protection of the password file.

[2] DES is slightly modified in crypt to prevent using an off-the-shelf DES chip to guess passwords.

3. Our Scheme

As advertized, our scheme is very simple. We deploy *two* salts, one public and one secret. The public salt is exactly the same as the current one. The secret salt is similar with one major difference: Like the password, but unlike the public salt, the secret salt is discarded by the system after use. It is not kept anywhere. (Unlike the password, it is not even kept by the user who does not need to know anything about it.) Like the public salt, the secret salt is generated at random at the time the password is first entered. We will evaluate the appropriate size of the secret salt when we analyze this scheme, but for the purpose of discussion let's assume that there are S possible values for secret salts (S will be in the 100-1000 range typically). The main idea behind this scheme is to select the value of S such that it will have a negligible effect on legitimate password validation and a major effect on password guessing.

In some sense the effect of the two salts is the same as one large salt, but there is one crucial difference. The original salt has no effect on the security of one password. It only makes it harder to guess many passwords at the same time. A secret salt directly makes each and every password much harder to guess. For each password guess, one will need to try S possible variations of the password, one for each possible value of the secret salt. A legitimate password validation process will have to do the same thing, but it allows very few attempts anyway. Making every attempt slower may not be a big problem if it is just slightly slower. And as it turns out by analyzing the parameters, it is possible to make it slower in an almost negligible way, and at the same time much harder to crack.

Let's assume that $S = 100$. In other words, the secret salt is chosen from a set of 100 possible values. The public salt remains the same at 12 bits, which implies 4096 possible values. Since these two salts are independent, there are 409,600 possible combinations of values for each single password. So, a list of all possible encrypted passwords, where the passwords are taken from a dictionary of size, say 100,000, would be of size 40,960,000,000. Is that large enough? That's hard to say, because it depends on the speed of each guess, which is why it is better to look at the analysis in terms of time rather than number of combinations.

Let's say that we allow the extra overhead for each login attempt to take half a second longer with this scheme than without it. Half a second is fast enough that most users will not be able to detect it at all. We can now figure out the value of S: It will be the number of encryptions that we can perform in half a second. That value depends, of course, on the computer and the encryption scheme. However, the protection that it provides can be evaluated independently: If each validation takes half a second, then trying all words in even a small dictionary, say of size 50,000 words, will take 7 hours. This is just for one password, not taking into account the public salt (which has no effect on a single password). Trying 100 passwords will take 700 hours due to the public salt, which still makes each password completely different. And this is for guessing extremely easy to guess passwords that come from a small dictionary. Trying a reasonable size dictionary of size 1 million will take about 6 days of CPU time just for one password. These are absolute numbers independent of the speed of encryption. Faster encryption will increase the number of guesses per minute, but will also increase the value of S to compensate for it. That means that only very weak passwords will have to be avoided. Almost everything else will be reasonably secure.

Of course, the guesser may have a more powerful faster computer or a bunch of them. But even if you reduce the time it takes to guess one password by an order of magnitude, it is still hours of CPU time, and this is for passwords that appear in the dictionary. Most computers today have speeds that fall within a range of one order of magnitude. (And even for those that are not, this scheme still offers a factor of S in the time it takes to guess.) Notice that the 20MB or so that was required to store all encrypted words in a dictionary grew to 20x4096 MBytes (= 80 GBytes) with the addition of the public salt, which is not that far from the realm of possibilities. This scheme makes it at least two orders of magnitude bigger.

We implemented this scheme within the UNIX login procedure, using the crypt procedure distributed with Crack [Mu92]. The value of S that corresponds to half a second slowdown of one login attempt was between 100 for an old Sequent machine to about 1000 for a DEC Alpha 400 (the slowest and fastest UNIX machines in our department). The implementation of this scheme is very easy. In particular, all changes are made as an extra filter to the login procedure, without having to modify the encryption mechanism at all. When a new password comes in, a random secret salt is generated in the range 1 to S. That number is then applied to the password to scramble it. It is not necessary to scramble in a strong cryptographic sense, because the result is going to be encrypted with the one-way function. The scrambling needs only to hash the given password into the set of all possible passwords to limit accidental duplicates. A good hash function will be sufficient. For example, we can use something like linear congruence, where the integer corresponding to the password is multiplied by the salt, and the remainer of dividing the result by a fixed large prime number is taken. This hashing takes a few instructions, so its running time is negligible compared to the encryption. The result is then sent to the usual one-way function procedure.

To verify a given password, the same process is applied, except that *all* secret salts in the range 1 to S are tried. In addition, we make sure that the original password is tried as well in case the password was entered before the new scheme took effect (with linear congruence, the original password corresponds to the secret salt of value 1). This way there is no need to change all passwords at the same time, or even distinguishing between old and new passwords. Furthermore, it is still possible to copy an encrypted password from one machine to another (which is a very useful procedure, for example, when setting up a new account) as long as the new machine is not significantly slower than the old one.

The value of S should be kept somewhere, but it is not crucial to make it secret. (It is crucial, of course, to protect that value against unauthorized changes, as it is crucial to so protect all encrypted passwords.) We recommend to set the value of S when a new machine is configured, and adapt it to that machine by following the ‘‘one validation in half a second’’ rule. The value of S can be raised at any time without touching any of the old passwords. This makes this scheme less dependent on current technology and in particular on the speed of the encryption. (For an added security feature, the secret salt can be chosen from the range B to $B+S-1$, with B being a large random number that is kept secret. But this makes the scheme harder to maintain.)

Notice that with our scheme, every password is translated into S different passwords, and any one of them is a valid password. Aren't we making passwords actually less protected, because some random passwords will now be accepted? Passwords in UNIX, which uses one DES block, are 56 bits long.

Therefore, there are about 72,000 *trillions* possible passwords. Dividing this number by *S*, which is, say 1000, will increase the probability of hitting a password at random by a factor of 1000, but it will still be about one in 72 trillions. Clearly an acceptable risk. Guessing a truly random password is unfeasible; the problem is with non-random passwords, which is what most people use. The main effect of this scheme is to make passwords much more random for everyone without having to remember random strings. Even with this scheme, weak passwords such as names or valid words should never be used, but the chances that a password can be quickly guessed become much much smaller.

Acknowledgements

Thanks to Gene Kim, Sean O'Malley, Hilarie Orman, Richard Schroeppel, and Gene Spafford for helpful discussions and comments, and to Phil Kaslo for doing most of the testing.

References

[Cu92] Curry D. A., *Unix System Security,* Addison-Wesley, Reading, Mass, 1992.

[GS91] Garfinkel S., and G. Spafford, *Practical UNIX Security*, O'Reilly & Associates, Inc., Sabastopol, CA, 1991.

[FK89] Feldmeier D. C., and P. R. Karn, ''UNIX password security — ten years later," *Proceedings of the UNIX Security Workshop* (August 1989).

[Ha91] *Icons of the computer age,* Harper's Magazine, November 1991, p. 26.

[JO89] Jobush D. L., and A. E. Oldehoeft, ''A Survey of Password Mechanisms: Weaknesses and Potential Improvements. Part 1,'' *Computers & Security*, **8** (July 1989), pp. 587−604.

[Kl90] Klein, D. V., ''Foiling the Cracker: A survey of, and improvement to, password security,'' *UNIX Security Workshop II,* the Usenix Association (August 1990), pp. 5−14.

[MW94] U. Manber and S. Wu, ''An Algorithm for Approximate Membership Checking With Application to Password Security,'' *Information Processing Letters* **50** (May 1994), pp. 191−197.

[MT79] Morris, R. and K. Thompson, ''Password security: a case history,'' *Communications of the ACM*, **22** (November 1979), pp. 594−597.

[Mu92] Muffett, D. E., ''Crack: A sensible password checker for Unix,'' a document distributed with the Crack 4.1 software package (March 1992).

[Sp92] Spafford, E. H., ''Opus: Preventing Weak Password Choices,'' *Computers & Security,* **11** (May 1992), pp. 273−278.

[St89] Stoll C., *The Cuckoo's Egg,* Doubleday, 1989.

[Wi68] Wilkes, M. V., *Time-Sharing Computer Systems,* American Elsevier, New York, 1968.