Protecting Secret Keys with Personal Entropy

Carl Ellison

Cybercash, Inc., cme@cybercash.com

Chris Hall

Counterpane Systems, hall@counterpane.com.

Randy Milbert

MIT and Counterpane Systems, rmilbert@mit.edu.

Bruce Schneier

Counterpane Systems, schneier@counterpane.com.

Abstract

Conventional encryption technology often requires users to protect a secret key by selecting a password or passphrase. While a good passphrase will only be known to the user, it also has the flaw that it must be remembered exactly in order to recover the secret key. As time passes, the ability to remember the passphrase fades and the user may eventually lose access to the secret key. We propose a scheme whereby a user can protect a secret key using the "personal entropy" in his own life, by encrypting the passphrase using the answers to several personal questions. We designed the scheme so the user can forget answers to a subset of the questions and still recover the secret key, while an attacker must learn the answer to a large subset of the questions in order to recover the secret key.

1 Introduction

> "Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations. (They are also large, expensive to maintain, difficult to manage, and they pollute the environment. It is astonishing that these devices continue to be manufactured and deployed. But they are sufficiently pervasive that we must design our protocols around their limitations.)" [9], p. 205.

In conventional encryption technology, users have one or more private keys (or other data) that they must keep secret. Usually the user's secret data is protected by encrypting it with a normal symmetric encryption algorithm using a key taken from the hash of a passphrase of the user's choice. However, if the user ever forgets the passphrase, then he or she loses access to the respective secret data. Alternatively, if the passphrase is short and simple enough to be remembered with certainty, then it is probably short and simple enough for an attacker to guess.

One solution is for the user to write the passphrase down and store it somewhere: e.g. a safe-deposit box. Another is for the system designer to eschew user-remembered passphrases entirely and store keys in physical tokens such as smart cards. Both of these solutions have problems.

We describe here a different alternative. Instead of a single passphrase, we employ multiple, simple passphrases. For analytical purposes, these can be thought of as a single, long passphrase with entropy that is the sum of the entropy of the individual passphrases.

To prevent loss of access by the user, through typing error or forgetting of some single passphrase, we apply fault tolerance techniques. In particular, we use secret sharing with parameters n and t, $t < n$, to split up the secret data into n parts. Each part is then encrypted separately, with a passphrase independent of all the others. If the user forgets the passphrases used to encrypt for up to $(n - t)$ of the parts, he can still recover the secret. Perhaps more importantly, the attacker must successfully guess t of these passphrases before being successful. If the passphrases are independent of one another and no individual decrypted share gives an appearance of being other than purely random, then the attacker's guesses amount to a brute force attack on a key space at least $(E_n t)$ in entropy, where E_n is the minimum entropy of the passphrases.

To remind the user of the small passphrases used, each encrypted share is related to a hint for its passphrase. These hints can be, for example, questions for which the answer is the passphrase. The list of hints can be determined ahead of time and used for all users or it can be uniquely created by each user. The latter option is more personal and likely to lead to stronger results, but in our experience it is very difficult for the novice user of this system to come up with such a list. It is also difficult for the developer to choose questions or hints for the general population that meet our requirements, but this is clearly added value from the user's point of view.

This system was first proposed in 1996[7] and has since been implemented. In this paper we outline the cryptography necessary to implement this scheme, describe the scheme we have implemented, and discuss the issue of choosing questions or hints.

The rest of the paper is organized as follows. In Section 2 we discuss the properties that a secret sharing scheme must have before we can apply it to our system. We describe the system that we built in Section 3. In Section 4 we describe the process of choosing n and t for the secret sharing scheme. Finally, in Section 5 we discuss the questions and answers used in the secret sharing scheme.

2 Secret Sharing Schemes

Unfortunately, not all secret sharing schemes are built alike. Aside from the obvious fact that not all secret sharing schemes are *ideal*, many have characteristics which are unsuitable for our application.

The following are our requirements for a secret sharing scheme:

(1) No individual decrypted share should give any information about whether the decryption key was correct. Otherwise, an attacker can do brute-force testing of each small passphrase separately.
(2) The minimum number of shares required to recover the secret must be protected with answers that have at least as much combined entropy as an adequate brute force work factor.

Shamir's Secret Sharing scheme based on Lagrange Interpolation over a finite field modulo a prime[11] is the classic example of an ideal secret sharing scheme. These scheme almost fits the bill for our system. Criterion 1 is almost met in Shamir's scheme and can be met exactly with a slight modification, described below. Criterion 2 is met by making the minimum threshold large enough. "Adequate" work factor is a matter of taste, although there is no reason to exceed the entropy of the secret being protected since in that case, the attacker can skip the shares and just do a brute force search for the secret directly.

2.1 Modification to Shamir for Criterion 1

Shamir's Secret Sharing[11] uses a polynomial

$$f(x) = \sum_{i=0}^{t-1} a_i x^i \bmod p$$

for which the user selects $(t-1)$ independent, random polynomial coefficients, a_i, inserts the secret to be shared as the constant coefficient, a_0, and evaluates the polynomial at n different points: e.g., $f(1), f(2), \ldots, f(n)$ to yield the n

shares. Under our scheme, each such share is then encrypted by a conventional encryption algorithm, yielding an encrypted share of b bits. In order not to lose information, $2^b > p$.

When a given encrypted share is decrypted by an attacker trying a brute force attack, the share will decrypt to a b bit value, v. If $v \geq p$, then the attacker knows that the decrypted value is not a correct share and therefore that the trial decryption key is incorrect. This has the effect of reducing the entropy of the passphrase used for that share by $(b - \log_2(p))$ bits.

For large p close to 2^b, especially for $p = 2^b - 1$, this entropy loss is very small. However, it is possible to modify the Shamir algorithm slightly so that there is no loss of passphrase entropy at all. To do this, we choose $p > 2^b$. This means that some shares of a given secret might exceed $(2^b - 1)$ and therefore not be expressable in b bits. However, this should happen with approximate probability

$$1 - \left(\frac{2^b}{p}\right)^n$$

if there are n shares being generated. If that probability is low enough, one can generate a polynomial at random, generate shares, test for shares exceeding $(2^b - 1)$ and if there are any, start over with a new random polynomial. The shares generated by this process will be uniformly distributed over the interval $[0..(2^b - 1)]$ and, as a result, the attacker will learn nothing from an individual trial share. In this case, the only test for correctness requires a full t shares. Again, the closer p is to 2^b, the better, but in this variant the distance from p to 2^b affects the process of creating the secret shares rather than providing a weakness for the attacker to exploit.

3 General Description of our System

The following is a description of the system for sharing and reconstructing the secret message. Here we assume that a legitimate set of questions and proper values for n (the number of questions) and t (the number of correct answers required to reconstruct the secret) have been chosen.

The system for sharing the secret message s is:

(1) Ask the user n questions $q_1, \ldots, q_n$ to generate answers $a_1, \ldots, a_n$.
(2) Generate a random number r_s to be used as salt.
(3) Compute the hash of each concatenation of question, answer, and random number: $h_1 = H(q_1 + a_1 + r_s), \ldots, h_n = H(q_n + a_n + r_s)$.

(4) Use a (n, t)-threshold scheme to split the secret message s into n secret shares $s_1, \ldots, s_n$.
(5) Encrypt each share using the corresponding hash as the key: $E_{h_1}(s_1) = c_1, \ldots, E_{h_n}(s_n) = c_n$.
(6) Keep $q_1, \ldots, q_n$, r_s, and $c_1, \ldots, c_n$ and discard the remaining data.

The salt value, r_s, is not needed for security. Even if the h_i is the same for two users, the s_i will be different and therefor the c_i will be different, eliminating the possibility of mounting a precomputed dictionary attack against the shares. Still, we feel it is safer to include a salt value. The system for reconstructing the secret message s is:

(1) Ask the user the same n questions $q_1, \ldots, q_n$ to generate answers $a'_1, \ldots, a'_n$.
(2) Compute the hash of each concatenation of question, answer, and random number: $h'_1 = H(q_1 + a'_1 + r_s)$, $\ldots$, $h'_n = H(q_n + a'_n + r_s)$.
(3) Decrypt each ciphertext using the corresponding hash as a key: $D_{h'_1}(c_1) = s'_1, \ldots, D_{h'_n}(c_n) = s'_n$.
(4) Using the (n, t)-threshold scheme, select subsets of t shares until the secret message s is correctly reconstructed. If at least t of the questions were answered correctly, the process will be successful. If fewer than t were answered correctly the user is unable to recover the secret. [1]

4 Choice of n and t

The user can choose n and t based on probabilities of correctness for individual answers. The user wants to guarantee success to himself or herself while prohibiting success by an attacker. In other words, the user needs to choose n and t so that the probability that the attacker will guess **as many as** t answers correctly is effectively zero while the probability that the user will guess **as few as** t answers correctly is effectively zero. Given such n and t, the attacker's best course is to choose the t answers with the lowest entropy and brute force them, while the user's best course is to answer all n questions and select subsets of size t at random until a correct subset is found. These processes are laid out in the sub-sections below.

[1] This is due to n being chosen too small. If the process for choosing n given in this paper had been used, then the user had over-estimated his per-question probability of success, P_0.

4.1 Choosing t

Suppose that the user picks n questions q_i such that the expected entropy for each answer is e_i. Note, different kinds of questions will have different values of e_i. A yes/no question will have e_i *at most* 1 whereas a question with a longer answer may very well have $e_i > 1$. In this section we assume that the user has appropriately assigned values e_i to each answer. We can assume without loss of generality that $e_i \leq e_j$ if $i < j$.

Under criterion 1 of Section 2, the best attack that an attacker can carry out against the secret sharing scheme is to brute force the minimum entropy set of t questions: $(q_1, \ldots, q_t)$. Assuming the questions are independent of one another, this vector of answers carries an entropy of

$$E(t) = \sum_{i=1}^{t} e_i \tag{1}$$

If the user wants an attacker to perform 2^{E^*} work, then t should be choosen such that

$$E(t) \geq E^* \tag{2}$$

Note, the larger that the user chooses t, the more work is required to recover the secret later on for both the user and the attacker. On the other hand, the extra work for the user is in the form of extra typing that is approximately linear in n (which must increase as t increases) while the work for the attacker is in the form of computation that is exponential in $E(t)$.

We assume the user will choose t as small as possible subject to (2). For example, if all questions were to be answered by first names (which have an entropy of approximately 8 bits) and if the user were to desire $E^* = 112$ bits [2], then $t = 14$.

4.2 Choosing n

If we assume that the legitimate user has the same probability, P_0, of answering each question correctly and that the correctness of each answer is independent, we can compute the probability, $P_1(k, n, P_0)$, that there will be exactly k

[2] the key size of 2-key 3-DES

correct answers out of n:

$$P_1(k, n, P_0) = \binom{n}{k} P_0^k (1 - P_0)^{n-k} \tag{3}$$

The probability that a user's given attempt to recover the secret will succeed, given a t out of n threshold scheme, is:

$$P_2(t, n, P_0) = \sum_{i=t}^{n} P_1(i, n, P_0) \tag{4}$$

P_0 needs to be measured by experiment for each given user. We assume the value 0.95 here, for lack of such experimental data over a large sample of users.

P_2 needs to be chosen by the user. It will be very close to one. To give guidance to the user in selecting the right value of P_2 and therefore of n given t, we might perform the following thought experiment. Assume one wants to recover this secret once a day for 100 years and have an expectation of only 1 failure in that time. This amounts to $M = 36525$ trials. In M trials, we expect $E_0 = M(1 - P_2)$ failures. Setting $E_0 = 1$ and $M = 36525$, we get $P_2 = 0.999972622$. From this desired value of P_2, we can then choose an acceptable region of (n, t) space.

For example, if we assume $P_0 = 0.95$ and $P_2 = 0.99998$ or higher, we find we can use (n,t) from the set:

n	t	n	t	n	t
5	1	6	1	7	1..2
8	1..3	9	1..3	10	1..4
11	1..5	12	1..6	13	1..7
14	1..7	15	1..8	16	1..9
17	1..10	18	1..11	19	1..11
20	1..12	21	1..13	22	1..14
23	1..15	24	1..16	25	1..17
26	1..17	27	1..18	28	1..19
29	1..20	30	1..21	etc.	etc.

From the previous section, we postulated a value of $t = 14$. Therefore, the user needs to choose $n \geq 22$ and we will assume the user chooses $(n, t) = (22, 14)$.

4.3 Expected workload for the user's software

Assuming $n = 22, t = 14, P_0 = 0.95$, a user might answer all 22 prompts but make some errors. The algorithm must then search for a correct subset of 14 shares to use to recover the secret. By criterion 1 of Section 2, the individual decrypted shares do not disclose whether they are correct, so the algorithm's only mechanism is to select subsets, reconstruct the secret and then try the resulting secret to see if it works.

If we select subsets of t shares at random, we can compute the expected number of subsets we would have to process before we would succeed at finding a correct subset. If we call each of those subset selections a trial, this yields the work load faced by the user's program measured in "trials".

If we assume there are exactly k correct shares out of n, the probability of selecting a correct subset of t shares is:

$$P_3(k, t, n) = \frac{\binom{k}{t}}{\binom{n}{t}} \tag{5}$$

If there are k correct out of n and we randomly choose subsets of size t, the expected number of trials until we find a correct subset is $P_3(k, t, n)^{-1}$. The probability that there are exactly k correct values is $P_1(k, n, P_0)$. Therefore, the expected number of trials we have to evaluate before succeeding is:

$$T = \sum_{i=t}^{n} P_1(i, n, P_0) P_3(i, t, n)^{-1} \tag{6}$$

if we select subsets at random from the n. For $n = 22, t = 14, P_0 = 0.95$, we get an expected number of trials, $T = 13.2$.

5 Choosing Questions

Choosing good questions is difficult but is probably the most important part of the system. We have developed three different mechanisms for choosing questions, but believe that this is an area ripe for future human-interface research.

5.1 General Questions

It is possible to generate questions that should apply to nearly everyone. For example:

(1) First song I danced to with an unrelated member of the opposite sex.
(2) First car I wished I could have owned.
(3) First vehicle I drove.
(4) Where I was during my first romantic kiss.

In the implementation we developed [3], the user was prompted with a series of questions like these. The user could either answer the question, in which case the answer was used to construct the secret, or not, in which case the question was ignored. This process continued until the user chose enough questions for the required entropy.

5.2 Custom questions/prompts

It is possible for users to generate prompts tailored to themselves, although this procedure adds difficulty. We have discovered that it takes considerable time to get into the right frame of mind to generate prompts. However, once in that frame of mind, it is possible to produce prompts at the rate of about one per minute. For example, one of the authors came up with the following prompts at that rate:

(1) response to the sentence: "I really like the clever way you ——."
(2) "(first) (last) (past) (prep) the train table."
(3) "(past) (first) (last) in the swimming pool"

These prompts have "answers" that make sense only to the person who created them, and can be much harder for an attacker to figure out than the answers to questions intended for everyone.

The trick is to select just enough of a reminder to trigger a strong memory in the user (preferrably from childhood, so that one knows the memory is long-term and fully established), but not enough of a reminder to give a private investigator anything to go on. The more misleading the prompt, the better.

The problem with this approach is that we have not yet discovered if every possible user is capable of getting into this frame of mind. Neither have we discovered a procedure for inducing that frame of mind.

[3] See `http://www.syncrypt.com`.

5.3 Free Association Prompts

It is conceivable to generate prompts as random collections of English words and let the user select those that mean something to him and skip those that don't. These have the advantage of being related to mental state rather than physical events and therefore of being stronger against private investigators. They have the disadvantage that they might not be related to a strongly emotional, long-term memory and therefore might not produce high probabilities of successful answer by the user. We have not done any research to determine the true entropy or reproducibility of such free association.

6 Conclusion

People are notoriously bad at remembering high-quality secrets over a long period of time. If a passphrase is long enough to provide adequate entropy for the underlying cryptographic key or other secret, it is hard to remember. As a result, users will either write the passphrase down, endangering security, or forget it completely, endangering their own access. Passphrases that are short or obvious enough to remember are often simple enough to be vulnerable to brute-force guessing attacks.

Our aim is to provide high-entropy passphrases without either of those compromises. Our tool gathers entropy in small pieces from a person's life experience. Our tool is fault-tolerant, so that the user can make the occasional mistake and still be able to recover his passphrase. At the same time, this fault-tolerance does not provide a comparable advantage to an attacker.

The following enhancements would make the system more robust.

- An application using this approach can quickly scan all decrypted shares, after a correct subset has been found, and test each one for correctness. With that data, an actual probability of success for each question can be accumulated. If there are some questions that are commonly incorrect, the user can be given the option to select a new prompt-answer pair to replace the forgotten one. By this process, the lifetime of the protected secret can be greatly increased.
- The statistics gathered as suggested above would yield an actual measurement of P_0, for use in future choices of (n, t).
- Individual question statistics should be able to guide a more efficient algorithm for choosing subsets.

Our work serves to demonstrate of the concept, but there is still much research

to be done with this idea.

- It is possible to find an ideal (minimum workload) search order for subsets of size t, for each (n, t), and prepare that subset list ahead of time. However, the obvious algorithm for this is extremely expensive and does not take advantage of any knowledge of different probabilities of success for different questions.
- We found that it takes a certain frame of mind to create custom questions or prompts, and that it was not easy to summon this frame of mind on demand. Even with practice, most of us found it very difficult to come up with good questions or prompts, and some people found it impossible. This has nothing to do with the cryptography (or even the computer engineering) of the system, and we expect that it is an interesting area of psychological research.
- Additional empirical research is required on the actual long-term memory of users. In our pilot system we assumed $P_0 = 0.95$, but with no more than a little empirical basis. Fortunately, running systems can gather that data as a side-effect and such applications might be modified to offer statistics to be submitted by e-mail or web to a common collection point.
- Research into the entropy of free association prompts should be done before that approach is seriously considered. However, once the entropy of free association is investigated, this may prove to be a valuable source of human-derived entropy for seeding random number sources. The user's ability to remember free association prompts also needs to be researched before this mechanism is considered for the algorithm described here.
- Research needs to be done on the actual entropy (from the attacker's point of view) of a given class of answers to normal questions or prompts. The only work we have done on the subject shows that first names in a moderate-sized community have an entropy of about 8 bits while full names have entropy in excess of 12 bits. However, we have not studied the size of an average user's personal address book (from which some answers might be chosen) or that of other pools of facts.

We believe that this application of personal entropy could have both frequent and infrequent uses. That is, an application like PGP, protecting a secret key with personal entropy, could employ this mechanism and the user would be providing answer sets quite frequently. An application like Password Safe [4], that backs up secrets for later emergency access, would have very infrequent uses. The user's probability of remembering answers, P_0, will probably differ in the two situations, leading to different values of n.

One might also need to provide other tools for the frequent user, to avoid user anger. That is, no matter how it is done, extra entropy requires extra

[4] `http://www.counterpane.com/passsafe.html`

keystrokes. The more keystrokes, the less user satisfaction we would expect. PGP, for example, allows a passphrase to be remembered by the system for a user-settable period of time. The more keystrokes in a passphrase, the longer that period of time is likely to be. The longer the period of time, the more likely a key (or compound passphrase) might be stolen.

To avoid user anger and still provide security, one might need to implement layered passphrases. That is, the high entropy compound passphrase could unlock a secret that is then encrypted under a low-entropy password (e.g., a 4-digit, randomly generated PIN). The user could then use that low-entropy passphrase until system shutdown or until failure to remember the PIN, at which time the PIN-encrypted secret would be deleted.

References

[1] G. S. Akl, P. D. Taylor, "Cryptographic Solution to a Multilevel Security Problem," *Proc. of CRYPTO '82*, Plenum Press, pp. 237–250.

[2] J. Benaloh, "Secret Sharing Homomorphisms: Keeping Shares of a Secret Secret," *Proc. of CRYPTO '86*, Springer-Verlag, pp. 251–260.

[3] J. Benaloh, J. Leicther, "Generalized Secret Sharing and Monotone Functions," *Proc. of CRYPTO '88*, pp. 27–35.

[4] G. Blackly, "Safeguarding Cryptographic Keys," *Proc. NCC*, Vol. 48, AFIP Press, 1979, pp. 313–317.

[5] E. F. Brickell, D. R. Stinson, "The Detection of Cheaters in Threshold Schemes," *Proc. of CRYPTO '88*, Springer-Verlag, pp. 564–577.

[6] B. Chor, S. Goldwasser, S. Micali, B. Awerbuch, "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults," *Proc. 26th IEE Symp. on Foundations of Computer Science*, 1985, pp. 383–395.

[7] C. Ellison, "Emergency Key Recovery without Third Parties," talk given at the Crypto '96 rump session, Aug 96.

[8] M. Ito, A. Saito, T. Nishizeki, "Secret Sharing Scheme Realizing General Access Structure," *Proc. Glob. Com(1987).*

[9] C. Kaufman, R. Perlman, M. Speciner, *Network Security: PRIVATE Communication in a PUBLIC World*, Prentice Hall, 1995, pp. 205.

[10] H. Lin, L. Harn, "A Generalized Secret Sharing Scheme with Cheater Detection," *Proc. of ASIACRYPT '91*, Lecture Notes in Computer Science 739, Springer-Verlag, pp. 149–158.

[11] A. Shamir, "How to Share a Secret," *Comm. of the ACM 22*, 11, Nov. 1979, pp. 612–613.

[12] M. Tompa, H. Woll, "How to share a secret with cheaters," Journal of Cryptology 1 (1988), pp. 133–138.

A Secret Sharing Schemes That Do Not Work

Our scheme essentially requires the reconstructor of the secret to differentiate between correct and incorrect shares. At first glance, secret sharing schemes with cheater detection seem ideally suited for this application. They do not work, however.

Cheater detection schemes (for example, [10]), which allow someone to detect a cheater with $k-2$ or fewer valid shares (assuming at least k shares are required to reconstruct the secret). An attacker can simply guess the answers used to encrypt the first $k-2$ shares, guess an answer used to encrypt the $k-1$ share, and apply cheater detection to determine whether or not the $k-1$st share is valid. Once the $k-1$st share is found, the kth share can be found using the same algorithm. The resulting entropy is

$$\sum_{i=1}^{k-2} E(a_i) + \max\{E(a_{k-1}), E(a_k)\} + 1$$

instead of

$$\sum_{i=1}^{k} E(a_i).$$

Most cheater detection schemes which allow one to detect cheaters without a k-set of valid shares allow pairwise cheater detection. Consequently the resulting entropy is at most

$$\min_i \{E(a_i)\} + \max_i \{E(a_i)\} + \log_2 k.$$

The scheme outlined in [10] is a perfect example.