User-Sure-and-Safe Key Retrieval

Daniel R. L. Brown*

April 28, 2008

Abstract

In a key retrieval scheme, a human user interacts with a client computer to retrieve a key. A scheme is user-sure if any adversary without access to the the user cannot distinguish the retrieved key from a random key. A scheme is user-safe if any adversary without access to the client's keys, or simultaneous user and client access, cannot exploit the user to distinguish the retrieved key from a random key. A multiple-round key retrieval scheme, where the user is given informative prompts to which the user responds, is proved to be user-sure and user-safe.

Remote key retrieval involves a keyless client and a remote, keyed server. User-sure and user-safe are defined similarly for remote key retrieval. The scheme is user-anonymous if the server cannot identify the user. A remote version of the multiple-round key retrieval scheme is proved to be user-sure, user-safe and user-anonymous.

Key Words: Key retrieval, User authentication, Passwords

1 Introduction

In a *key retrieval* scheme, a human user interacts with a set of computers to recover a cryptographic key. Only the authorized user should be able to retrieve the key. A scheme is *local* or *remote* depending on whether the set of computers is a single local computer or an open network of computers, respectively.

A key retrieval scheme is *user-sure* if the retrieved key is indistinguishable from a random key for any adversary that does not interact with the user; for example, a corrupted client is such an adversary. Exhaustive search (including dictionary) attacks are thereby excluded against user-sure key retrieval scheme. Simple password-encrypted keys are not user-sure. Corrupted clients can have access to the password-encrypted key and can launch a dictionary attack on the password. Salt values and deliberately slow functions, such as those used in PKCS #5, are generally used to slow down such attacks, but the resulting system is not fully user-sure. Canetti, Halevi and Steiner [1], and Ford and Kaliski [3] describe user-sure key retrieval schemes, local and remote, respectively. The former supplements passwords with human only solvable puzzles.

A (remote) key retrieval scheme is *user-safe*, if the retrieved key is indistinguishable from a random key for any adversary that does not have access to the client (server) key, and that also cannot interact with the user and client (server) simultaneously. A user-safe scheme requires the user to act upon on a failed key retrieval session(s). The maximum number of such failures without

*Certicom Research

complete compromise of the retrieved key is the *tolerance*. Password-encrypted keys and other password-only schemes, such as [3] are not user-safe, because the attacker needs only a single access to the user to capture the user password. (They have zero tolerance.) The scheme [1] requires the user to enter a password and to solve some human-only solvable puzzles, so could potentially be user-safe, but only with tolerance of one, because the password can be obtained with one failed session, after which the human-only solvable puzzles can be obtained from the client.

This paper proposes a user-sure key retrieval scheme with a user-safe tolerance of up to about eight. It is a multi-round protocol, where in each round, the user is given a *prompt*, and makes a *selection* (compare to [4]). Subsequent prompts depend on previous user selections. The user aborts when confronted with an invalid prompt, primarily because the correct selection will not be evident. The remote version is also *user-anonymous* (as is [3]) in the sense that an honest client makes it impossible for the server to identify the user.

2 Proposed Key Retrieval Scheme

There are two versions of the protocol, a local version and a remote version. We first describe the local version. We then describe the remote version as a modification of the local protocol.

2.1 Local Key Retrieval

An overview of the local key retrieval scheme is given in Figure 1. There are n rounds. In the i^{th} round, the user is given a prompt p_i from the client, which is computed by a function Client described below. Then, the user makes a selection u_i, using a simple function User. If user does not

$$
\begin{array}{lcl}
\underline{\text{User}\qquad\qquad\qquad} & & \underline{\text{Client}\qquad\qquad\qquad\qquad} \\
 & & p_1 = \mathsf{Client}() \\
 & \xleftarrow{p_1} & \\
u_1 = \mathsf{User}(p_1) & \xrightarrow{u_1} & p_2 = \mathsf{Client}(u_1) \\
 & \xleftarrow{p_2} & \\
u_2 = \mathsf{User}(p_2) & \xrightarrow{u_2} & p_3 = \mathsf{Client}(u_1, u_2) \\
 & \xleftarrow{p_3} & \\
\vdots & \vdots & \vdots \\
u_{n-1} = \mathsf{User}(p_{n-1}) & \xrightarrow{u_{n-1}} & p_n = \mathsf{Client}(u_1, \ldots, u_{n-1}) \\
 & \xleftarrow{p_n} & \\
u_n = \mathsf{User}(p_n) & \xrightarrow{u_n} & k = F(u_1, \ldots, u_n)
\end{array}
$$

Figure 1: Overview of Local Key Retrieval

get the correct prompt p_i in round i, then the user aborts the session. The user is to discontinue completely after a few aborted sessions, and seek to re-initialize the system. If the client does not get the correct user response u_i in round i, it continues as if everything is fine. Indeed, the client has no information available to know whether the individual user selections u_i are correct. After the final user input u_n, the client device computes the *retrieved key* as

$$k = F(u_1, \ldots, u_n). \tag{1}$$

Only once k is derived, is the client permitted to determine whether the user has made the correct selections.

Preliminary experiments with users have demonstrated the number of possible $(u_1, \ldots, u_n)$ user secrets can actually exceed the number that is usually ascribed to conventional passwords. Nevertheless, a human has limitations, so the number of possible user secrets cannot be arbitrarily high. Well-known techniques of *salting* and *deliberately slow functions* can be applied to the retrieved key k.

2.1.1 The User Function

Function User takes as input a prompt value p. This may be something displayed to the user on the monitor, for example. The output of User is either to abort, which we indicate by $\perp$, or to reveal a secret value. Precisely, function User is specified by the values $(p_1, \ldots, p_n)$ and $(u_1, \ldots, u_n)$, as follows:

$$\mathsf{User}(p) = \begin{cases} u_i & \text{if } p = p_i, \\ \perp & \text{otherwise.} \end{cases} \tag{2}$$

The *user secret* is $(u_1, \ldots, u_n)$, and it will be assigned to the user uniformly at random. The exact values of $(p_1, \ldots, p_n)$ and $(u_1, \ldots, u_n)$ are determined in conjunction with the client algorithm to be described below.

For algorithm User to be implemented by a human user, the number of possible values for each u_i must generally be quite small. Typically, we make the numbers the same for each u_i. For example, we may choose $c = 16$ possible choices for each u_i. If we also have $n = 16$, then the number of possible user secrets is $c^n = 16^{16} = 2^{64}$. (The successful preliminary user experiments used these settings.)

The number of possible prompts can be made considerably larger than 2^{64}. Human-computer user interfaces are generally such that human can take in more information than they can put out. In particular, the prompts can depend on quite large cryptographic values, which they will, in this protocol. However, humans are not good at processing purely random looking data, so to form the prompts p_i from random looking cryptographic values, redundancy must be added. This redundancy will cue the human user to respond correctly. This is achieved by a *prompt* function implemented by the Client function, as described below.

2.1.2 The Client Function

Algorithm Client takes input $(u_1, \ldots, u_{i-1})$, for $i \geq 1$, where the case $i = 1$ indicates an empty input. The client computes the prompt value

$$p_i = P(F(u_1, \ldots, u_{i-1})) \tag{3}$$

where F is a strong *cryptographic function*, such as a (keyed) hash function, and P is a *prompt* function that renders arbitrary bit strings into a more user-friendly format whereby a user will be consistently able to respond with a value u_i.

The security properties needed for the function F will become evident after the security analysis. We assume that range of F has size f, and that its domain includes at least the set of possible secrets $(u_1, \ldots, u_n)$ and possibly bit strings of any length. (Typically, we will have $f = 2^s$, with the outputs of F being bit strings of a fixed length s.) The primary property needed by the prompt function P is to assist the user to reliably implement the User function. See §A for a discussion prompt functions. This paper analyzes the protocol under the assumption that the user will be able to compute User and to recognize that the p_i are in the correct order. This paper will not focus on the properties of P needed to make this possible.

2.1.3 Initialization of the Protocol

The user selections $u_1, \ldots, u_n$ are assigned uniformly at random[1] to the user. Let the total number of choices for $(u_1, \ldots, u_n)$ be u. Preliminary experiments have shown successful user selection with $u = 2^{64}$.

The user must be trained to learn the prompts $p_1, \ldots, p_n$ and the responses $u_1, \ldots, u_n$. This learning process can take the form of several practice executions of the scheme. In the practice mode, further hints may be given to the user. Some preliminary tests have shown that such training can lead to quite good recall, even after a delay of months.

2.2 Remote Key Retrieval

An overview of the proposed remote key retrieval scheme is given in Figure 2. The scheme is a modification of the local key retrieval scheme. From the user's perspective, there is no difference between the local and remote key retrieval schemes. Computation of function Client has now been split between a local client and a remote server. Loosely speaking, we may think of the server

$$
\begin{array}{lcrlcl}
\text{User} & & & \text{Client} & & \text{Server} \\
\hline
 & & (b_1, c_1) = & \mathsf{Blind}() & \xrightarrow{c_1} & s_1 = \mathsf{Server}(c_1) \\
 & \xleftarrow{p_1} & p_1 = & \mathsf{Unblind}(b_1, s_1) & \xleftarrow{s_1} & \\
u_1 = \mathsf{User}(p_1) & \xrightarrow{u_1} & (b_2, c_2) = & \mathsf{Blind}(u_1) & \xrightarrow{c_2} & s_2 = \mathsf{Server}(c_2) \\
 & \xleftarrow{p_2} & p_2 = & \mathsf{Unblind}(b_2, s_2) & \xleftarrow{s_2} & \\
u_2 = \mathsf{User}(p_2) & \xrightarrow{u_2} & (b_3, c_3) = & \mathsf{Blind}(u_1, u_2) & \xrightarrow{c_3} & s_3 = \mathsf{Server}(c_3) \\
 & \xleftarrow{p_3} & p_3 = & \mathsf{Unblind}(b_3, s_3) & \xleftarrow{s_3} & \\
\vdots & \vdots & \vdots & & \vdots & \vdots \\
u_{n-1} = \mathsf{User}(p_{n-1}) & \xrightarrow{u_{n-1}} & (b_n, c_n) = & \mathsf{Blind}(u_1, \ldots, u_{n-1}) & \xrightarrow{c_n} & s_n = \mathsf{Server}(c_n) \\
 & \xleftarrow{p_n} & p_n = & \mathsf{Unblind}(b_n, s_n) & \xleftarrow{s_n} & \\
u_n = \mathsf{User}(p_n) & \xrightarrow{u_n} & k = & H(u_1, \ldots, u_n) & &
\end{array}
$$

Figure 2: Overview of Remote Key Retrieval

having computed a keyed part of function F, while client has computed an unkeyed part, and we write

$$F = H \circ U \circ S \circ B \circ M \circ H \tag{4}$$

[1] As an alternative, it is possible for users to choose the values $u_1, \ldots, u_n$, for example, by making u_1 their favorite response to p_1, and u_2 their favorite response to p_2 and so on. This may aid the user to more easily implement User. This, however, introduces the serious risk that the number of possible user secrets $(u_1, \ldots, u_n)$ is harmfully reduced to the point of being exhaustible by a concerted attack. In any case, it certainly takes away control of the entropy of the user secret from the cryptographic system. We will not analyze this alternative approach to initialization.

where H, U, B, M are functions computed by the client and S is a function computed by the server. The function H is essentially a hash function. The function M is a massaging function used to minimally modify the output of H to make it suitable for application of the function B. The pair of functions (B, U) are selected at random, but such that they obey the relationship that $U \circ S \circ B = S$. The function B is a blinding function, one of whose effects is to shield the user's anonymity. The function S is a keyed function, to prevent adversaries from easily presenting malicious prompts to the user. The function U is unblinding function. For B to be an effective blinding function, it must be a probabilistic function. The function F will be a deterministic function because U will undo the randomization of B. A deterministic F is crucial for the user must always see the same prompt p_i.

In terms of Figure 2, we have $\mathsf{Server} = S$ and $\mathsf{Blind} = B^* \circ M \circ H$, and $\mathsf{Unblind} = P \circ H \circ U^*$, where B^* outputs both the random blinding factor b_n and the blinded value obtained by applying B, while U^* takes as input the blinding factor b_n and unblinds the server-computed value.

The way we ensure $U \circ S \circ B = S$, is to take $U = B^{-1}$ and to have $S \circ B = B \circ S$. Commuting functions may be realized based on Diffie-Hellman key exchange. For simplicity, we use the additive notation common in an elliptic curve cryptography. Function S can be defined as $S(X) = sX$, where s is some secret value held by the server. Function B can be defined as $B(X) = bX$ for some randomly selected with b. Of course, then, $B^{-1}(X) = b^{-1}X$, where the b^{-1} is short for $b^{-1} \bmod q$ where q is the order of the elliptic curve group used. We may write $B^*(X) = (b, bX)$ where b is selected at random, and we write $U^*(b, Y) = b^{-1}Y$.

The massaging function M, here, could take a pseudorandom bit string given by hash function H, and represent this as a potential x-coordinate of an elliptic curve point. If this x-coordinate does not correspond to a valid a field element or to a point on the elliptic curve, then it can be incremented. Any bias that may introduce the resulting point on the elliptic curve is likely irrelevant, because the blinding function B will produce an unbiased point.

3 Security Definitions for Key Retrieval

When formally defining adversaries corresponding to the user-sure and user-safe security objectives, we take the approach from [1] of requiring the adversary to distinguish the retrieved key from a random key. What this ensures is that any use of the retrieved key, such as to encrypt data, is as good as the same thing with a random key.

With this definition, it appears that one could take any key retrieval scheme and make it secure as follows. Pick a random key k', and encrypt it with the retrieved key k. We may write the encrypted random key suggestively as $k(k')$. The new key retrieval scheme stores $k(k')$ on the client, and the retrieved key is defined to be k', is retrieved by first retrieving k through the old scheme, and then decrypting $k(k')$ using k. Clearly, k' is a random key, so it is indistinguishable from one. (Thus this seems secure.) This construction will fail, however, because the adversaries that we will consider are permitted either access to the client secrets or access to the user. In the former, for example, the adversary gets $k(k')$. It is not clear that if the adversary could distinguish original retrieved key k from a random key, then the adversary could not distinguish the modified retrieved key k' from a random key, once it is given the additional information in the form $k'(k)$.

3.1 User-Sure Local Key Retrieval

Informally, a local key retrieval scheme is *user-sure* if it requires an adversary to have access to the user in order to have any ability to distinguish the retrieved key from random.

Definition 1. A *user-free* adversary A to a local key retrieval scheme must distinguish between the retrieved key k and a random value (key) from the range of the function F. The user-free adversary gets all information available to the client, including any secret keys. The user-free adversary does not get to interact with the user. We may quantify A by its advantage in distinguishing the key from random and by its running time.

This definition is similar to the one in [1], except that we do not include the human only solvable puzzles and human oracles to solve them.

Definition 2. A *user-sure* local key retrieval scheme has no user-free adversary with reasonable advantage and running time. We may quantify the degree of user-sure security by the running time and advantage of the user-free adversaries.

3.2 User-Safe Local Key Retrieval

Informally, a local key retrieval scheme is *user-safe* if an adversary with a limited access to the user, and non-simultaneous access to an honest client, cannot distinguish the retrieved key from random. In other words, the user cannot be fooled into revealing the user secret to an unauthorized client.

Definition 3. A *user-fool* adversary A to a local key retrieval scheme must distinguish between the retrieved key and a random key from the range of F. Adversary A does not get access to the client secret. Adversary A may participate in q_U sessions with the user and q_C session with the client, but these session must not be simultaneous. At the end of each client session, the adversary gets access to the retrieved key resulting from the session.

The user-fool adversary can model attacks on passwords in which an adversary sets up a front site where users enter passwords. If a user uses a common password for all sites, which is common despite usually being against a password policy, the user-fool adversary will succeed. Even if a user tries to uses a different password for each site, to err is human, and the password of a legitimate site may accidentally be entered into the adversary's site. Even a careful user may be fooled if the adversary creates a site that looks like a legitimate site.

Simultaneous user and client access is not allowed[2] for a user-fool adversary A. Our local key retrieval scheme, and perhaps any other purely cryptographic scheme, cannot stop an adversary with simultaneous access. Some kind of physical countermeasures seem necessary to stop such an adversary.

Definition 4. A *user-safe* local key retrieval scheme is one where no user-fool adversary exists with reasonable running time and advantage.

[2] A simultaneous attack also includes the passive observation of the u_i or even the prompts p_i during a session between the user and an honest client, such as an *over-the-shoulder* attack.

3.3 User-Sure and User-Anonymous Remote Key Retrieval

Informally, a remote key retrieval is *user-sure* if an adversary cannot distinguish the retrieved from a random key, even given the server secret key and indirect access to the user via an honest client.

Definition 5. A *user-mediated* adversary A must distinguish between a retrieved key and a random key, with reasonable running time and advantage. The adversary has access to the server secret key. The adversary may interact with the user through an honest client.

A user-mediated adversary A may be thought of as a corrupt server. In some respects, it is like the user-free adversary, with the addition that it gets indirect access to the user through the honest client.

Definition 6. A *user-sure* remote key retrieval scheme has no user-mediated adversary with reasonable running time and advantage.

Similarly, we may consider the possibility of a server identifying the user.

Definition 7. A *user-identifying* adversary A_I must distinguish between interaction the user through an honest client and interaction with a randomized client. Adversary A_I may have access to the server secret key.

Sometimes, it is desirable to avoid user-identifying adversaries.

Definition 8. A *user-anonymous* remote key retrieval scheme has no user-identifying adversary, with reasonable running time and advantage.

3.4 User-Safe Remote Key Retrieval

Informally, a remote key retrieval scheme is *user-safe* if an adversary that interacts both with the user and with an honest server, but not both simultaneously, cannot distinguish the retrieved key from random.

Definition 9. A *user-fool* adversary A of a remote key retrieval scheme can directly interact with the user in at most q_U sessions and with an honest server in at most q_S session, but cannot interact with both the user and the server simultaneously.

Remote and local user-fool adversaries are similar, except that in the remote case, F is now implemented as $F = H \circ S \circ M \circ H$, and the adversary basically gets free access to the function S, which is not a conventional pseudorandom function, because it commutes with blinding functions.

Definition 10. A *user-safe* remote key retrieval scheme has no user-fool adversary with reasonable running time and advantage.

We will need a security assumption about S. We use a *one-more evaluation* assumption. Let $N \geq 1$ be an integer. Let B_N be an adversary to S, indexed by N. Adversary B_N is given oracle access to S, and is allowed to make N queries. After this, it is given a random point X, and either $Z = S(X)$ or a random point $Z = Y$. Algorithm B_N must distinguish whether $Z = S(X)$ or $Z = Y$. For $N = 1$, this is essentially the usual decisional Diffie-Hellman problem: given $(G, S(G), X, Z)$, determine whether $Z = S(X)$ or Z is random. We say that S is (N, σ)-secure if any B_N algorithm with advantage σ is infeasible.

4 Security Results for the Propose Schemes

This section states the security results. More technical proofs of the results are relegated to §B.

4.1 User-Sureness of the Local Scheme

We use the notation A^F to indicate that adversary A with the particular choice of the function F. The goal is for $A^F(z)$ to output 1 if it thinks that z is the user's retrieved key, or 0 if it thinks that z is a random element from the range of F.

Lemma 1. *Suppose that, for F a random oracle, a user-free adversary A^F can distinguish the key $k = F(u_1, \ldots, u_n)$ for a random user secret $(u_1, \ldots, u_n)$, from a random bit string r, making at most q queries to random oracle F, and with advantage $\alpha = \Pr\left(A^F(k) = 1\right) - \Pr\left(A^F(r) = 1\right)$. Then we have*

$$\alpha \leq \frac{q}{u}\left(1 - \frac{1}{f}\right)^q, \tag{5}$$

where the probabilities are defined over the random choices of both F and of A^F.

Qualitatively, this lemma is obvious: the output of a random oracle on an input with a moderate amount of randomness is difficult to distinguish from fully random values in the range of the random oracle. Less obvious, perhaps, is quantification of the probability of distinguishing such outputs.

The random oracle model is often referred to as a *non-standard assumption*, because in practice no real function is a random oracle. It is natural, therefore, to seek a weaker assumption about F. One such assumptions is that F is a *pseudorandom generator*. This means that F takes a small random input and produces an output that appears to be large random output. In other words, this assumption on F is precisely that the conclusion of Lemma 1 for the specific choice of F. A proof is therefore unnecessary.

Theorem 1. *If F is a pseudorandom generator with respect to the space of user secrets $(u_1, \ldots, u_n)$, then the conclusions of Lemma 1 hold, except that q is not the number of oracle queries, but rather a term in the computational cost of user-free adversary A.*

While this seems to be just assuming what one wants to prove, we justify it by the fact that the pseudorandom generator assumption is an accepted one, for certain choices of F. For example, pseudorandom generators F have been constructed based on one-way functions.

4.2 User-Safeness of the Local Scheme

As before, a superscript F in A^F indicates running the given adversary A using the particular function F. In particular, A has access to F through the client. Also, we will consider an algorithm B that attempts to distinguish between a pseudorandom function F and a random function R. We write B^F and B^R to indicate that B has access to an oracle for computing F or R, respectively. For any oracle function G, we write $B^G()$ to indicate its output, as it has no input other than the oracle.

Theorem 2. *Suppose that F is a pseudorandom function with respect to the key space of user secret values $(u_1, \ldots, u_n)$. Suppose R is a random function containing in its domain the set of user secrets $(u_1, \ldots, u_n)$ and having a range identical to F. Suppose that $k = F(u_1, \ldots, u_n)$ for some random user secret. Suppose that A is a user-fool adversary with advantage $\alpha = \Pr\left(A^F(k)\right) - \Pr\left(A^F(r)\right)$. Let q_C be the number of protocol executions that A_U makes to the client oracle. Let q_U be the number of protocol executions made with the user. Then, provided that $q_U < n$, one can construct an algorithm B with advantage of distinguishing F from R with advantage*

$$\beta = \Pr\left(B^F() = 1\right) - \Pr\left(B^R() = 1\right) \geq \frac{\alpha}{2} - \frac{q_C - q_U}{2c^{n-q_U}} \tag{6}$$

where B makes at most $1 + (n+1)(q_C + q_U)$ queries to the challenge function oracle (whether pseudorandom or random).

What this theorem means is that, provided $q_C \ll q_U + c^{n-q_U}$ and F is a pseudorandom function, then user-fool adversary A will fail. In practice, users should learn to abort failed sessions, and to react anything more than a few failures, effectively putting a limit on q_U for the adversary A. On its part, the client should react to a large number of failed sessions, putting a limit on q_C for the adversary. Together, the client and user can thwart the adversary A.

4.3 User-Sureness-And-Anonymity of the Remote Scheme

An honest client has the power to blind the corrupt server. This blinding is information-theoretic, that is, it is perfect blinding. Therefore, the adversary A_S learns nothing about the user secret u_i, no matter how much computation it does. With no information about the user secret, the adversary gains no information about k, and cannot distinguish it from random value, provided that H can act as a good pseudorandom generator. Thus we have proven:

Theorem 3. *If H is a pseudorandom generator on the space of user secrets, then adversary A_S cannot distinguish the retrieved key from a random key.*

The client does not send any user-specific information to the adversary, and blinding is perfect so we clearly have:

Theorem 4. *The remote key retrieval scheme is user-anonymous, even given unlimited running time.*

Anonymity may not always be desirable. In such cases, the system may be set up so that the client forwards user identity to the server.

4.4 User-Safeness of the Remote Scheme

To prove things gradually, we begin a result where H is a random oracle. This is a weak result, because it makes a strong assumption. We write q_S for the number of sessions with the server, and q_U for the number of sessions with the user. We write q_H for the number of queries to the random oracle hash made by A^H. As usual we write $k = F(u_1, \ldots, u_n) = H(S(M(H(u_1, \ldots, u_n))))$ for some random user secret $(u_1, \ldots, u_n)$, and we write r for some random value in the range of H (which, is also the range of F). As usual, we use superscript to indicate oracle dependence, so we write $A^{H,S}$ to indicate that A is run with oracles $H()$ and $S()$.

Lemma 2. *Suppose that H is a random oracle. Suppose that S is (N, σ)-secure. Suppose that A^H is successful remote user-fool adversary, with advantage $\alpha = \Pr\left(A^H(k) = 1\right) - \Pr\left(A^H(r) = 1\right)$, that uses q_S server sessions and q_U user sessions. Then ...*

Proof. The main idea of the proof is that the adversary can only succeed if also defeats the security of S. The main difficulty in proving this is that the user does not weaken the system. We apply the same argument as in the proof of Theorem 2 to argue that user does not weaken the system. $\square$

Next, we argue that if H is replaced by a function that is a pseudorandom generator on both the space of user secrets and the group of elliptic curve points, then it security result holds.

Theorem 5. *If H is a pseudorandom generator as above, and S is secure as above, then no remote user-fool adversary A exists.*

As a final caution, care should be taken with regard to the security of the function S. Results such as Cheon's [2] suggest that for large N, the computational work needed for a successful distinguishing is less than the usual square root of the group size. Even if a carefully chosen group can escape this phenomenon, this should still be a consideration.

References

[1] R. Canetti, S. Halevi, and M. Steiner. Mitigating dictionary attacks on password-protected local storage. In C. Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes Ccomputer Science*, pages 160–179. Springer, Aug. 2006.

[2] J. H. Cheon. Security analysis of the strong Diffie-Hellman problem. In S. Vaudenay, editor, *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *Lecture Notes Ccomputer Science*, pages 1–11. Springer, Apr. 2006.

[3] W. Ford and B. Kaliski. Server-assisted generation of a strong secret from a password. In *IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, NIST, Gaithersburg MD, June 2000.

[4] M. Jakobsson and S. Myers. Delayed password disclosure. *International Journal of Applied Cryptography*, 1(1):47–59, 2008.

A Prompt Function Examples

Suppose that F is based on a 256-bit hash function, such as SHA-256. Take a dictionary of 2^{16} words. Divide the output of F into 16 segments of 16 bits. Convert each segment into an integer with which one can index the dictionary. Form a list of sixteen words corresponding from these 16 indices. The defines the prompt function P.

In this example, the user response to a prompt is a 4 bit value, which is the index of the list of the sixteen words. To input this choice, the user may click on the word. The user, therefore, remembers the user inputs $u_1, \ldots, u_n$ as words.

With this particular prompt function example, we note two points. First, occasionally, a list of sixteen words may contain the same word twice. In this case, either the user must be able to remember the choice by its position, or we may always assign the minimum index of any matching words. The former is more difficult for the user, while the latter slightly reduces the number of possible user secrets. It is also possible to avoid repeated list entries with a better encoding algorithm (such as arithmetic encoding).

Secondly, we generally presume that the user will abort if prompt p_i is not as exactly as expected. In practice, however, human users may actually have some error rate in accepting incorrect p_i, especially if they look almost correct. With the example prompt function, this could occur if the human users sees the secret word in the list, but the other words are missing. We will provide a security analysis for this kind of error-prone user in a separate security analysis.

Other prompt functions are possible. For example, the prompt can be formed by making an image derived from the cryptographic value. The user then selects are part of this picture. Alternatively, a database of existing images of can be used instead of generated.

B Technical Proofs

B.1 Proof of Lemma 1

Proof of Lemma 1. Suppose that input to A^F is z, and we want to analyze what the output of $A^F(z)$ is, depending on whether $z = k = F(u_1, \ldots, u_n)$ where $(u_1, \ldots, u_n)$ is chosen uniformly at random from u possible values, or $z = r$ where is chosen uniformly at random as some bit string of the same length as k.

Adversary A^F can make up to q queries to the random oracle F. It does not hurt the advantage of F to make q distinct queries, so we will assume this. Let Q be the set of inputs that A^F makes to oracle F. Let Z be the set of outputs of oracle F to the q queries made. Note, that we do not assume that F is injective, so Z could have size less than q. In fact, this is likely. In any case, Z has size at most q.

Suppose that $z = k = F(u_1, \ldots, u_n)$. There are u values for $(u_1, \ldots, u_n)$. The probability that it is one of the q distinct queries made by A^F is exactly q/u, because A^F has no access to the user, and thus no information about $(u_1, \ldots, u_n)$. (Note that k, being the output of random oracle, leaks no information about $(u_1, \ldots, u_n)$.) If $(u_1, \ldots, u_n) \in Q$, then clearly $z = k \in Z = F(Q)$. The probability that the user secret is not one of the q queries is A^F is $1 - \frac{q}{u}$. In this case, $z \notin Z$ with probability $(1 - 1/f)^q$, where the probability here includes the choices of F. (The conditional probability, after the q choices have been made by F, is generally greater, since it is $(1 - 1/f)^{|Z|}$.) Otherwise, $z \in Z$. So for $z = k$, the overall probability that $z \in Z$ is:

$$\Pr\left(A^F(k) = 1\right) = \frac{q}{u} + \left(1 - \frac{q}{u}\right)\left(1 - \left(1 - \frac{1}{f}\right)^q\right). \tag{7}$$

(If we condition on the q choices of F, then this is a lower bound.) Now suppose that $z = r$. The probability that $z \notin Z$ is just $(1 - 1/f)^q$, so the probability that $z \in Z$ is just $\Pr\left(A^F(r) = 1\right) = 1 - (1 - 1/f)^q$.

Given the probabilities above, we claim that the optimal strategy for A^F is

$$A^F(z) = \begin{cases} 1 & \text{if } z \in Z, \\ 0 & \text{if } z \notin Z. \end{cases} \tag{8}$$

To see this, we first note that the advantage, as we have expressed it, namely $\alpha = \Pr\left(A^F(k) = 1\right) - \Pr\left(A^F(r) = 1\right)$, has an alternative form. Let b be selected at uniform from $\{0, 1\}$. Let z_1 have the distribution of $k = F(u_1, \ldots, u_n)$ for uniformly random user secret, and let z_0 have the distribution of r, namely uniformly at random from the range of F. Let $\chi(x) = 1$ if 1 if $x = 0$ and otherwise let $\chi(x) = -1$. Then $\alpha = E(\chi(A^F(z_b) - b))$. What this means that k or r are equally likely to be fed to A^F, the probability that it guesses the correct choice minus the probability is incorrect equals the advantage. This may be thought of a game based definition of advantage.

Now $z \in Z$ is more likely to have occurred when $b = 1$, and $z \notin Z$ is more likely to have occurred when $b = 0$. Therefore (8) represents a maximum likelihood strategy based on whether or not $z \in Z$. Finally, because F is a random oracle and r is uniform and $(u_1, \ldots, u_n)$ is uniform, the only useful information provided by z and Z to A^F about the bit b is the bit of information whether or not $z \in Z$.

Now we just subtract $\Pr\left(A^F(r) = 1\right) = (1 - (1/f))^q$ from the expression (7) for $A^F(k)$, and after some cancellations we get (5). $\square$

B.2 Proof of Theorem 2

Proof of Theorem 2. The main idea is to construct B via a reduction that uses A as a subroutine. Suppose that $G = F$ or $G = R$, and the challenge of B^G is determine which is the case. To do this, we use A^G. If A^G is successful in distinguishing between $k = G(u_1, \ldots, u_n)$ and r, then B^G reports that $G = F$. Otherwise, B^G reports that $G = R$. The only difficulty in the details of this construction are analyzing the advantage β of B in terms of q_C and q_U. We assume that A outputs 0 or 1, with $A(z) = 1$ indicating that it is saying that it thinks that its the challenge input key is the real retrieved key k.

Algorithm B chooses a random $b \in \{0, 1\}$ and a random user secret $(u_1, \ldots, u_n)$. If $b = 0$, then it chooses random $z_0 = r$ in the range of F (which is the same as the range of F and of R). If $b = 1$, then B computes $z_1 = k = G(u_1, \ldots, u_n)$, using a call to its oracle function G. Now B^G runs $A^G(z_b)$. Algorithm A^G expects to have a client oracle, which, on input $(u_1, \ldots, u_{i-1})$, outputs $p_i = R(G(u_1, \ldots, u_{i-1})$. To answer the queries of A_G to the client oracle, algorithm B calls its oracle function G on input $(u_1, \ldots, u_{i-1})$ to compute $g_i = G(u_1, \ldots, u_{i-1})$. Now B computes $p_i = R(g_i)$ as the client response.

When A executes the protocol with a user, algorithm B^G, responds with the correct u_i provided that A^G supplies the correct prompt. Otherwise reduction B^G aborts the execution of the protocol, just as a user should do. To test the correctness of the prompt, reduction B^G again calls its G oracle to compute $p_i = P(G(u_1, \ldots, u_{i-1})$.

As before, let χ be the characteristic function of zero: $\chi(x) = 1$ if $x = 0$ and otherwise $\chi(x) = -1$. The output of B^G may then be expressed as $\frac{1}{2}(1 + \chi(A^G(z_b) - b))$.

That the reduction algorithm B makes at most $1 + (n + 1)(q_C + q_U)$ queries to G can be seen as follows. One query may be made if $b = 1$. For each of the q_C interactions with client, there are n prompts which must be computed using the oracle G. Also, there is one retrieved key per client session. For each of the q_U sessions with the user, there are n prompts which must be computed using the oracle G. Of course, if the inputs of any of these G-queries are identical, then B^G can use fewer calls to the G oracle. The point is, however, that B^G uses at most $1 + (n + 1)(q_C + q_U)$ queries to G.

It is clear that

$$\Pr\left(B^G() = 1\right) = E(B^G()) = \tfrac{1}{2}(1 + E(\chi(A^G(z_b) - b))). \tag{9}$$

Therefore the advantage of B is

$$\beta = \Pr\left(B^F() = 1\right) - \Pr\left(B^R() = 1\right) = \tfrac{1}{2}\big(E(\chi(A^F(z_b) - b)) - E(\chi(A^R(z_b) - b))\big) \tag{10}$$

As we noted earlier, an alternative formulation of advantage of A^F is that $\alpha = E(\chi(A^F(z_b) - b)$. We claim that $E(\chi(A^R(z_b) - b)) = \gamma$ where γ is the probability that A queries $(u_1, \ldots, u_n)$ to the client and will be computed later. This gives

$$\beta = \frac{\alpha - \gamma}{2} \tag{11}$$

To establish this claim of expectation γ, we argue that the case of $b = 0$ is equivalent to a modified oracle function R', where $R'(u) = R(u)$ except if $u = (u_1, \ldots, u_n)$, in which case $R'(u) = z_0$ instead of z_1. Because z_0 is selected by B completely at random, the function R' will actually be a full random function. No algorithm, including A, can distinguish R from R' unless it queries $(u_1, \ldots, u_n)$ to its oracle. By the same token, we define $z'_0 = z_1$ and $z'_1 = z_0$, and $b' = 1 - b$. Note that $z_b = z'_{b'}$. Then, provided A does not query $(u_1, \ldots, u_n)$, we have

$$E(\chi(A^R(z_b) - b)) = E(\chi(A^R(z'_{b'}) - b)) = E(\chi(A^{R'}(z'_{b'}) - b)) = E(\chi(A^R(z_b) - b')) = -E(\chi(A^R(z_b) - b)). \tag{12}$$

The second last equation follows because (R, b, z_b) has the same distribution as $(R', b', z'_{b'})$. This proves that $E(\chi(A^R(z_b) - b) = 0$, if A does not query $(u_1, \ldots, u_n)$.

It remains only to assess the probability γ that A queries $(u_1, \ldots, u_n)$ to its equivalent of the R oracle. For each protocol session with the client, adversary A gets one opportunity to query a value that could be a full user secret. We assert that the adversary can learn at most the first q_U entries in the user the secret, but otherwise gets no other information about the user secret. (This assertion is to be proven later below.) From the adversary's perspective, this then means that the number of possible user secrets is

$$c^{n - q_U} \tag{13}$$

Essentially, the adversary must devote q_U sessions with the client to learn the first q_U entries of the user secret. The $q_C - q_U$ remaining sessions with the client can be used to make guesses at the user secret. The probability of hitting the true user secret is

$$\gamma = \frac{q_C - q_U}{c^{n - q_U}} \tag{14}$$

Therefore, when the adversary queries the correct user secret $(u_1, \ldots, u_n)$ it will immediately detect whether the resulting key matches the challenge, and correctly guess b, so $\chi(A^R(z_b) - b) = 1$, and $E(\chi(A^R(z_b) - b)) = \gamma$.

The argument above hinges on the assertion that q_U sessions with the user will only allow the adversary to learn the first q_U entries in the user secret. We now prove this assertion. This depends on the user aborting if any prompt value is wrong, including if they appear in the wrong order. In particular, the user will not respond with u_i, unless the user has already provided the previous

$u_1, \ldots, u_{i-1}$ in the current session. So during any session with the adversary, the user will reveal some initial segment of the user secret and nothing else.

We use induction on q_U. The base case of $q_U = 0$ is obvious, because the adversary gets no information about the user secret from the user. Suppose $q_U = m \geq 1$, and our assertion is true for $q_U = m - 1$. Then m sessions with the user, reveal at most $(u_1, \ldots, u_m)$ and nothing else about the user secret. Given $m + 1$ sessions, the first $m - 1$ sessions can only reveal at most $(u_1, \ldots, u_{m-1})$. In the m^{th} session, the adversary can send $(u_1, \ldots, u_{m-1})$ to the client and obtain p_m, which can be sent to the user. The user will respond with u_m, so the adversary can learn at least u_m.

What we need to show, however, that the adversary cannot learn anything beyond u_m, in the m^{th} session. By the argument about initial segments of the user secret, it suffices to show that nothing can be learned about u_{m+1}. The only way to learn anything about u_{m+1} is to correctly guess p_{m+1} during the current session with the user. The adversary is not allowed to query the client during the user session. The adversary has negligible chance of guessing p_{m+1}, at least out of the blue.

On the other hand, the adversary could have queried the client before, using each of the possible c values of u_{m+1}, in c separate client sessions. However, in each session, the client must also have known u_m which was only learned with certainty during the current user session, making for c^2 previous client sessions. Even with this information, the adversary has only a $\frac{1}{c}$ of guessing the correct value of p_{m+1} to which the user will reveal u_{m+1}. Such advance sessions to the client provide the adversary no more advantage than the client sessions after learning an initial portion of the user secret. $\square$