A Note on Session Key Distribution Using Smart Cards

Victor Shoup

Bellcore, 445 South St., Morristown, NJ 07960

shoup@bellcore.com

July 24, 1996

Abstract

In this note, we discuss some variants of the session-key protocol of Shoup and Rubin presented at Eurocrypt '96, and we discuss some implementation issues.

1 Introduction

Shoup and Rubin (Eurocrypt '96) propose a protocol for session key distribution in a setting where "smart cards" are used to hold the participants' long-term keys. In this note, we make some clarifications and correct some minor errors; we discuss variants that are more efficient, while still preserving security; and we discuss some implementation issues.

2 Random nonces versus counters

In §2.2 of Shoup and Rubin, it is suggested that the strings r and s in Protocol SK1 could be replaced by counters. This is indeed true. However, the claim in §3.2 that the same holds for Protocol SK3 is simply not true. If one traces through the proof, one sees that the randomness of r is required to "sandwich" the smart-card query $C_j(2, i, r)$ between the two message-queries made to host A. Indeed, if r were predictable, the adversary could make the query $C_j(2, i, r)$ at any time, and then at some later time, it could correctly respond to A's challenge without accessing the smart card.

Thus, r must be a random string; however, a careful examination of the proof does show that s may be a counter.

3 Re-using the same key

It is possible to use a single key K_i instead of 3 keys in SK1 and the 4 keys in SK3. To do this, one uses the simple trick of making the function calls independent by partitioning the inputs using "pad strings." The details will be presented below.

4 Eliminating a smart card query

The query $C_j(4, i, s, \delta)$ can be eliminated altogether by having $C_j(2, i, r)$ simply output δ itself. The comparison can be done by the host.

5 The new protocol SK3

Using the above observations, we describe the new Protocol SK3. The key server S has a single secret key K, from which it can compute the long-term key $K(i)$ of host i as $f_K(i)$. On input (i, j), the key server outputs

$$\epsilon = f_{K(i)}(01 \cdot j) \oplus f_{K(j)}(00 \cdot i),$$

and

$$\alpha = f_{K(i)}(10 \cdot \epsilon \cdot j).$$

There are three types of queries that can be made of all smart cards, where we write $C_i(\cdot)$ for a query to i's smart card. For clarity, we sometime use host i and sometimes j.

- $C_i(1) = (r, f_{K(i)}(11 \cdot r))$, where $r \leftarrow R_k$.
- $C_j(2, i, r) = (s, \beta, \sigma, \delta)$, where $s \leftarrow R_k$, $\kappa = f_{K(j)}(00 \cdot i)$, $\beta = f_\kappa(1 \cdot r \cdot s)$, $\sigma = f_\kappa(00 \cdot s)$, $\delta = f_\kappa(01 \cdot s)$.
- $C_i(3, j, r, s, \epsilon, \alpha, \beta, \gamma) = (\delta, \sigma)$. This is computed as follows. Check if $f_{K(i)}(10 \cdot \epsilon \cdot j) = \alpha$. If not, output "bad α" and quit. Check if $f_{K(i)}(11 \cdot r) = \gamma$. If not, output "bad γ" and quit. Set $\kappa = \epsilon \oplus f_{K(i)}(01 \cdot j)$. Check if $f_\kappa(1 \cdot r \cdot s) = \beta$. If not, output "bad β" and quit. Set $\delta = f_\kappa(01 \cdot s)$ and $\sigma = f_\kappa(00 \cdot s)$ and output (δ, ω).

The process-to-process protocol is as follows. We assume process A is an initiator on host i, and process B is a responder on host j. Upon acceptance, a process assigns the session key to the variable σ.

Step 1a A sends (i, j) to S.

Step 1b A sets $(r, \gamma) = C_i(1)$ and sends r to B.

Step 2a S receives (i, j) from A and sends the corresponding ϵ and α to A.

Step 2b B receives r from A, sets $(s, \beta, \sigma, \delta) = C_j(2, i, r)$, and sends s, β to A.

Step 3 A receives ϵ, α from S and s, β from B. A computes $C_i(3, j, r, s, \epsilon, \alpha, \beta, \gamma)$. If this is an error value, A rejects, otherwise A accepts, assigns this value to (δ', σ), and sends δ' to B.

Step 4 B receives δ' from A, and accepts if $\delta = \delta'$, and rejects otherwise.

6 Implementation Issues

6.1 Binding Names to Keys

In the above description of Protocol SK3, we have not specified what we mean by a name.

There are several possibilities. One is to view a name in the protocol as a small, unique code number. Securely binding a code number to a socially-meaningful "identity" then requires some other mechanism, such as public-key signatures. In this case, it is important that the signing authority never allows the same identity to get bound twice, since otherwise imitation attacks would be possible. This requires that a database of used identities is maintained.

Another possibility is to simply use self-identifying names. This avoids the need for a public-key infrastructure. However, a database of used names must also be maintained in this case, not only to prevent imitation attacks, but also to prevent private keys from being revealed. To see this, consider the following scenario. When a smart card is manufactured, it is given a special initial key which it effectively shares (via a pseudo-random function) with the device G that generates personal long-term keys. Given a name, G computes an the corresponding long-term key and outputs an authenticated encryption of this key, using the initial key of the given smart card. Now, provided no smart card is ever broken (obtaining its initial key), this scheme is only susceptible to imitation attacks (assuming G maintains no database of names). However, if an attacker should break *any* card, obtaining its initial key, then by gaining access to G (by bribery, break-in, etc.) the long-term key associated with *any* name can be obtained.

This type of attack is entirely unacceptable. It can be defeated if G maintains a database of names, which defeats imitation attacks as well.

For reasons of efficiency, "real world" names can be hashed using a hash function like MD5 or SHA. If this hash function is collision free, one can always use the hash of a name instead of the name itself in protocol SK3. In particular, the hashing can be done on the host, and not on the card itself—this can greatly reduce the card's I/O and computation.

6.2 Choice of Pseudo-random function

One could use a hash function like MD5 or SHA with a secret key, using the secret key as part of the message block or as part of the initial vector. Given an implementation of such a hash function, implementing SK3 should be easy.

Another choice is to use DES, using DES in CBC mode for longer inputs. However, one must be very careful doing this, as CBC-DES is far from a perfect pseudo-random functions; it can be subject to simple "cut-and-paste" attacks and to certain types of "birthday" attacks.

We describe now an implementation of Protocol SK3 using DES. This implementation assumes names are 128-bit hash-values (or truncated hash values). We write $h_1(i)$ and $h_2(i)$ for the two 64-bit blocks of the hash of name i. All the hashing can be done by the hosts, and only hash-values are sent to the smart cards.

We write $f_k(x)$ to denote DES applied to a single 64-bit block x with key k. We write $f_k(x_1, \ldots, x_m)$ to denote m blocks processed in CBC mode, i.e., $f_k(x_1, \ldots, x_n) = f_k(f_k(x_1, \ldots, x_{m-1}) \oplus x_m)$ for $m > 1$. Note that implementations of CBC-DES that always use an initial vector should simply be programmed to use x_1 as the initial vector.

The long-term key $K(i)$ associated with the name i is $f_K(h_1(i), h_2(i))$, where K is a master key.

A secret key $K(i)$ is issued by a key issuing server G, which contains K. On input i, G checks if key $K(i)$ has already been issued, and if not, computes $f_K(h_1(i), h_2(i))$. Because of weaknesses in CBC-DES, it is important that G compute the hash of i itself. As mentioned above, G may output an authenticated encryption of $K(i)$ under an initial card key.

The key server S also contains the secret key K. On input (i_1, i_2, j_1, j_2), S computes $K_1 = f_K(i_1, i_2)$ and $K_2 = f_K(j_1, j_2)$, and outputs

$$\epsilon = f_{K_1}(1, j_1, j_2) \oplus f_{K_2}(0, i_1, i_2),$$

and

$$\alpha = f_{K_1}(2, \epsilon, j_1, j_2).$$

Because of the sensitivity of the master key K, one might choose to use triple-DES, which doubles the length of the key K. We assume in this implementation that single-DES is used everywhere else.

We now describe the three types of smart card queries. Each card maintains a single counter I_i, which is restricted to 62 bits.

- $C_i(1) = (r, r', \gamma)$, where $r \leftarrow I_i$, $I_i \leftarrow I_i + 1$, $r' = f_{K(i)}(10 \cdot r)$, and $\gamma = f_{K(i)}(11 \cdot r)$.
- $C_j(2, i_1, i_2, r, r') = (s, \beta, \sigma, \delta)$, where $s \leftarrow I_j$, $I_j \leftarrow I_j + 1$, $\kappa = f_{K(j)}(0, i_1, i_2)$, $\beta = f_\kappa(10 \cdot s, r, r')$, $\sigma = f_\kappa(00 \cdot s)$, $\delta = f_\kappa(01 \cdot s)$.
- $C_i(3, j_1, j_2, r, r', s, \epsilon, \alpha, \beta, \gamma) = (\delta, \sigma)$. This is computed as follows. Check if $f_{K(i)}(2, \epsilon, j_1, j_2) = \alpha$. If not, output "bad α" and quit. Check if $f_{K(i)}(11 \cdot r) = \gamma$. If not, output "bad γ" and quit. Set $\kappa = \epsilon \oplus f_{K(i)}(1, j_1, j_2)$. Check if $f_\kappa(10 \cdot s, r, r') = \beta$. If not, output "bad β" and quit. Set $\delta = f_\kappa(01 \cdot s)$ and $\sigma = f_\kappa(00 \cdot s)$ and output (δ, ω).

The process-to-process protocol is as follows. We assume process A is an initiator on host i, and process B is a responder on host j. Upon acceptance, a process assigns the session key to the variable σ.

Step 1a A sends $(h_1(i), h_2(i), h_1(j), h_2(j))$ to S.

Step 1b A sets $(r, r', \gamma) = C_i(1)$ and sends r, r' to B.

Step 2a S receives (i_1, i_2, j_1, j_2) from A and sends the corresponding ϵ and α to A.

Step 2b B receives r, r' from A, sets $(s, \beta, \sigma, \delta) = C_j(2, h_1(i), h_2(i), r, r')$, and sends s, β to A.

Step 3 A receives ϵ, α from S and s, β from B. A computes $C_i(3, h_1(j), h_2(j), r, r', s, \epsilon, \alpha, \beta, \gamma)$. If this is an error value, A rejects, otherwise A accepts, assigns this value to (δ', σ), and sends δ' to B.

Step 4 B receives δ' from A, and accepts if $\delta = \delta'$, and rejects otherwise.

That completes the description of the protocol. It has been carefully designed to resist various types of cut-and-paste and birthday attacks. The precise ordering of the blocks in the CBC computations, and the funny 0's and 1's, are all critical.

We note that in the type 3 card query, the value r' could be computed by the card itself (from r). This might save on space and card I/O, but requires an extra DES evaluation. This does not appear to be necessary to avoid obvious attacks on CBC-DES, but it can't hurt.

One might also let the host in step 1a send i and j to S and let it compute the hashes of these names itself. This would increase the load on the server. This does not appear to be necessary Again, this does not appear to be necessary to avoid obvious attacks on CBC-DES, but it can't hurt.