SRP #11:
Secure Remote Passwords behind PKCS #11

Rick van Rein
OpenFortress

November 29, 2015

Abstract

Secure Remote Passwords provide a zero-knowledge proof system for password-protected service access; furthermore, the protocol establishes a session key with forward secrecy properties. These properties make it a desirable alternative to many other password schemes; its integration as a TLS cipher suite also makes it practically usable. In this paper, we find what is required to use SRP with a randomly generated password that is protected by a PKCS #11 token.

1 Introduction

The SRP formalism is related to Diffie-Hellman, but it is not the same. This means that implementation on a PKCS #11 token, with its stringent assumptions on the use of DH and the accompanying protection of secrets, is not trivial. In fact, as will be shown here, we need to modify the client side of the protocol to be able to use SRP. Since only the client-side changes and the cryptographic properties of the protocol are not impacted, this is a reasonable choice to consider.

Formalism of SRP. The following exchange is defined for SRP. Client and service are assumed to agree on DH parameters, a generator g and a hash function H. All equalities in this paper are modulo the DH modulus. The password is fixated by the client as P, which may be shared among many services without loss of security. For each service, the client generates a random salt s and passes it with a verifier v to the service while setting up an account C. So, the service stores $C \mapsto \langle s, v \rangle$ and treats v as a secret, because it might elicit password cracking attempts. The client stores C, which it sends to the service to get back the salt s that it setup in the past.

Client	**Protocol**	**Service**

$$
\begin{array}{rcl}
x = H(s, P) & & \\
v = g^x & & \\
 & C, s, v \longrightarrow & \text{store } C \mapsto \langle s, v \rangle
\end{array}
$$

During authentication, the client picks random value a and the service picks random values u and b. We assume the entropy of each of these three values to match the size of the to-be-derived session key. The authentication protocol is conducted as follows:

$$
\begin{array}{rcl}
\textbf{Client} & \textbf{Protocol} & \textbf{Service} \\
 & C \longrightarrow & \text{lookup } \langle s, v \rangle \\
 & \longleftarrow s & \\
x = H(s, P) & & \\
A = g^a & & \\
 & A \longrightarrow & \\
k = H(N, g) & & k = H(N, g) \\
 & & B = k \cdot v + g^b \\
 & \longleftarrow B & \\
u = H(A, B) & & u = H(A, B) \\
S = (B - k \cdot g^x)^{a+ux} & & S = (A \cdot v^u)b \\
K = H(S) & & K = H(S) \\
M_1 = H(A, B, K) & & \\
 & M_1 \longrightarrow & \text{verify } M_1 \\
 & & M_2 = H(A, M_1, K) \\
\text{verify } M_2 & \longleftarrow M_2 &
\end{array}
$$

Rules of the game. We define the following constraints to the work laid out in this paper:

- The purpose of using PKCS #11 is to protect credentials from extraction, so that the SRP functions of the client cannot be implemented without the PKCS #11 token holding the password.
- The random value a is not protected by PKCS #11 because it only impacts the session key, and this value is also known to the service.
- Intermediate values will be analysed; they are considered 'public', that is need no PKCS #11 protection, when they can be derived on the service-side of the protocol. Note however, that a passive observer sees less.

- We only define client-side computations with PKCS #11, because the the service side does not handle any secrets, thanks to the zero-proof nature of the protocol.
- We follow the assumption of SRP that the verifier is kept a secret by the service side; this is chiefly in the interest of the service, the client could employ SRP salt pinning to enforce it. Salt pinning falls outside the scope of this paper.

2 First Naive Implementation

To demonstrate the problems that arise when implementing unchanged SRP on a PKCS #11 interface, we start off by making a sound technical implementation, and point out where this fails to satisfy the desired password protection for which PKCS #11 is being considered.

Password establishment. A new password is generated with the DH mechanism in C_GenerateKeyPair, in such a way that it must remain on the token. This will be the password P for the formalism. Since the client performing this computation is also the party to generate salt and verifier, and to authenticate on later returning, there should be no conflict of interest to support key export. Only for reasons of backup might wrapped export be permissible.

Credentials generation. To derive service access credentials from the password, first generate a random salt s. The value is public, and may be generated on or off the PKCS #11 token. Then, the token's C_DigestXXX functions must be used to compute the hash value $x = H(s, P)$ on the token; the password can be incorporated with C_DigestKey pointing at P.

Now run C_DeriveKey with g as the base value and point to x as the private key or exponent. The outcome is $v = g^x$ which is supplied in a pair $\langle s, v \rangle$ to the service, together with a username.

Authentication. The client starts off as basic DH with C_GenerateKeyPair. The private key will be known as a and the public key is A. Send A over to the service.

The service generates $B = v + g^b$ for a local random secret b and sends B and u to the client, where u is another random scrambling parameter that is publicly shown.

The client now computes $B - g^x = B - v$, which is entirely based on public values. The result equals g^b, which can be combined with session key a in C_DeriveKey to find g^{ab}, a normal DH result.

Then, B^u is computed from public values, and used as a base key in C_DeriveKey, with DH private key x. This yields $B^{ux} = g^{bux}$, a value that can also be calculated separately on the service, as $v^{bu} = g^{xbu}$.

The values g^{ab} and g^{bux} are now multiplied outside of PKCS #11 to find shared secret S, which is of course also derived on the service. Likewise, the session key $K = H(S)$ is also available to the service and not considered a secret that would benefit from PKCS #11 protection.

Problem Analysis. The value of this implementation is that it never needs to extract the password P, so PKCS #11 can be exploited as a barrier to the secret. However, this implementation calculates x from a C_DigestFinal invocation, which means it is delivered into a byte array that is extracted from PKCS #11. The value x is the only part of the protocol that contains P, so knowing x is as useful as knowing P, at least for a service that uses the given salt value s.

In fact, in terms of the original AKA formalism of which SRP is an instance, the computation of x falls outside the formalism as a mere parameter. The use of x in the SRP protocol may be considered a technical aspect, rather than a cryptographic necessity. This means that we may consider modifications to the calculation of x to make it better suited for PKCS #11 implementation.

3 Client-side Variation on the Formalism

The proposed variation to better adapt SRP to PKCS #11 is to use another one-way function than $H(s, P)$. Within the SRP formalism, the operations for the DH mechanism offers an alternative, namely the PKCS #11 function C_DigestKey which can be used as a modular-exponentiation operation, possibly incorporating secret exponentials.

As an alternative to x in plain SRP, we define $x' = H(s)P$ to improve the integration. This looks awful, because it is possible to derive P from the values of x' and s, but the damage is controlled by not actually depending on the value x' in the implementation; instead, we use it to modify the formalism. Note that the hash function has no cryptographic role in the AKA formalism, so we might even have specified sP instead of $H(s)P$. We choose to retain the hash because the salt may be used to carry additional values to pin down the service, and that could be computationally expensive or run into size constraints without the hash function.

During secret establishment, the client does not generate a random P on the token, but instead constructs a DH key pair $\langle p, P \rangle$:

$$p = g^P$$

Both P and p are concealed by PKCS #11, with no permission of extracting them. Wrapped export might be supported to facilitate backup and recovery, but the values of p and P are assumed to not appear in plaintext outside of PKCS #11.

When setting up a service with a verifier and salt, the following process is used:

$$\begin{array}{rcl}
\textbf{Client} & \textbf{Protocol} & \textbf{Service} \\
x' = H(s)P & & \\
v' = p^{H(s)} & & \\
 & C, s, v' \longrightarrow & \text{store } C \mapsto \langle s, v' \rangle
\end{array}$$

The service cannot tell the difference, because the values s is treated as an opaque string and because it is not of a different format; the value v' is also not distinguishable for the service.

During authentication, the client picks random numbers a and the service picks random values u and b. The authentication protocol is conducted as follows:

$$\begin{array}{rcl}
\textbf{Client} & \textbf{Protocol} & \textbf{Service} \\
 & C \longrightarrow & \text{lookup } \langle s, v' \rangle \\
 & \longleftarrow s & \\
v' = p^{H(s)} & & \\
A = g^a & & \\
 & A \longrightarrow & \\
k = H(N, g) & & k = H(N, g) \\
 & & B = k \cdot v' + g^b \\
 & \longleftarrow B & \\
u = H(A, B) & & u = H(A, B) \\
S' = (B - k \cdot v')^a \cdot (((B - v')^u)^{H(s)})^P & & S = (A \cdot v'^u)^b \\
K' = H(S') & & K = H(S) \\
M_1' = H(A, B, K') & & \\
 & M_1' \longrightarrow & \text{verify } M_1' \\
 & & M_2' = H(A, M_1', K) \\
\text{verify } M_2' & \longleftarrow M_2' &
\end{array}$$

The calculation procedure for S' by the client differs from the original calculation, but ought to yield the same output as the standard calculation of S by the service under proper authentication conditions. The same then holds for K', M'_1 and M'_2. We only use the accents to the symbols to distinguish their calculation and be able to prove equality.

Note that the value x' is never seen in this calculation, nor can it be derived. The value P is used, but cannot be extracted from PKCS #11.

In terms of the original work on SRP, this is still an AKA formalism. The change from $x = H(s, P)$ to $x' = H(s)P$ occurs in the periphery surrounding the AKA formalism definitions. We will consider efficiency and security impacts in subsequent sections, but first turn to an implementation in terms of PKCS #11, using standard DH mechanisms.

Proof of correctness. During authentication, the two sides need to find the same values for S and S', after which the values for K', M'_1 and M'_2 follow without a change. The thing to prove is that S' in the client-side formalism yields the same result as S in the service-side formalism, given that the service has received the value v' instead of v. The correctness is proven as follows:

$$
\begin{aligned}
S' &= (B - k \cdot v')^a \cdot (((B - k \cdot v')^u)^{H(s)})^P \\
&= (g^b)^a \cdot (g^b)^{uH(s)P} \\
&= g^{ba} \cdot g^{buH(s)P} \\
&= g^{ab} \cdot ((g^P)^{H(s)})^{ub} \\
&= (g^a)^b \cdot (p^{H(s)})^{ub} \\
&= A^b \cdot v'^{ub} \\
&= (A \cdot v'^u)^b \\
&= S
\end{aligned}
$$

This establishes that the unmodified SRP service algorithm, as well as the unmodified networking protocol, can work with the modified client that this formalism introduces. The one thing to care for is that all clients adhere to the same new formalism, since they will need to base their work on v' and not v. Since the password pair $\langle p, P \rangle$ is concealed by PKCS #11, this constraint is already implied by the technology.

Problem Analysis. The formalism presented here suffers from a cryptographic problem, namely that the value p is assumed to be secret. In normal Diffie-Hellman operations, only P would be considered secret, and p would be a public key. Indeed, the value of p could be recomputed from g and P by anyone with PKCS #11 access, and extracted. The impact of this extraction would be that new values of v' could be calculated without the help of PKCS #11; only authentication would continue to rely on live access to

PKCS #11, but that is not the best possible result. We therefore construct one further iteration.

Another problem that remains is the potential relationship between values of v. Consider two values of $H(s)$, called h_1 and h_2. Since these are random values, it may happen that h_2 is divisable by h_1, so when $v_1 = p^{h_1}$ then $v_2 = p^{h_2} = v_1^{h_2/h_1}$. This problem is mitigated in plain SRP through the use of the hash function, and the next iteration must also resolve it through better scattering of the verifiers.

4 Maximally PKCS #11 Protected Formalism

On top of the suitability for PKCS #11 in previous variation, the additional requirement is now to avoid that verifiers can be constructed from extractable values like p. This is ensured by applying the secret key P to the salt.

As an alternative to x and x', we therefore define $x'' = H(s)^P P$. This introduces a multiplication with secret P, which will make it impossible to derive password-equivalent value x'' in the implementation. It also raises the salt to the power P to require the password before the verifier can be computed. Under the assumption that services conceal the verifier, a new service cannot be setup with the same salt, and the dependency on P implies that only the client with PKCS #11 access can thus create a new salt and verifier for a new service.

The secret key P established as before, with token-generated random key P, protected from extraction. But in this variation, the value p does not have to be protected from extraction.

$$p \quad = \quad g^P$$

When setting up a service with a verifier and salt, the following process is used:

Client	**Protocol**	**Service**
$x'' = H(s)^P P$		
$v'' = p^{H(s)^P}$		
	$C, s, v'' \longrightarrow$	store $C \mapsto \langle s, v'' \rangle$

As before, the service cannot tell the difference, because the value s is treated as an opaque string and because it is not of a different format; the value v'' is also not distinguishable for the service.

During authentication, the client picks random value a and the service picks random values u and b. The authentication protocol is conducted as follows:

Client	**Protocol**	**Service**
	$C \longrightarrow$	lookup $\langle s, v'' \rangle$
	$\longleftarrow s$	
$v'' = p^{H(s)^P}$		
$A = g^a$		
	$A \longrightarrow$	
$k = H(N, g)$		$k = H(N, g)$
		$B = k \cdot v'' + g^b$
	$\longleftarrow B$	
$u = H(A, B)$		$u = H(A, B)$
$S'' = (B - k \cdot v'')^a \cdot (((B - k \cdot v'')^u)^{H(s)^P})^P$		$S = (A \cdot v''^u)^b$
$K'' = H(S'')$		$K = H(S)$
$M_1'' = H(A, B, K'')$		
	$M_1'' \longrightarrow$	verify M_1''
		$M_2'' = H(A, M_1'', K)$
verify M_2'';	$\longleftarrow M_2''$	

The calculation procedure for S'' by the client differs from the original calculation, but ought to yield the same output as the standard calculation of S by the service under proper authentication conditions. The same then holds for K'', M_1'' and M_2''. We only use the accents to the symbols to distinguish their calculation and be able to prove equality.

Note that the value x'' is never seen in this calculation, nor can it be derived. The value P is used, but cannot be extracted from PKCS #11.

In terms of the original work on SRP, this is still an AKA formalism. The change from $x = H(s, P)$ to $x'' = H(s)^P P$ occurs in the periphery surrounding the AKA formalism definitions. We will consider efficiency and security impacts in subsequent sections, but first turn to an implementation in terms of PKCS #11, using standard DH mechanisms.

Proof of correctness. During authentication, the two sides need to find the same values for S and S'', after which the values for K'', M_1'' and M_2'' follow without a change. The thing to prove is that S'' in the client-side formalism yields the same result as S in the service-side formalism, given that the service has received the value v'' instead of v. The correctness is proven as follows:

$$S'' \quad = \quad (B - k \cdot v'')^a \cdot (((B - k \cdot v'')^u)^{H(s)^P})^P$$

$$
\begin{aligned}
&= (g^b)^a \cdot (g^b)^{uH(s)^P P} \\
&= g^{ba} \cdot g^{buH(s)^P P} \\
&= g^{ab} \cdot ((g^P)^{H(s)^P})^{ub} \\
&= (g^a)^b \cdot (p^{H(s)^P})^{ub} \\
&= A^b \cdot v''^{ub} \\
&= (A \cdot v''^u)^b \\
&= S
\end{aligned}
$$

This establishes that the unmodified SRP service algorithm, as well as the unmodified networking protocol, can work with the modified client that this formalism introduces. The one thing to care for is that all clients adhere to the same new formalism, since they will need to base their work on v'' and not v or v'. Since the password P is concealed by PKCS #11, this constraint is already implied by the technology.

5 Implementation as SRP #11

The following describes how to implement the last iteration of the SRP mechanism and demonstrates its integration with PKCS #11. Where the distinction is useful, we will distinghuish 'plain SRP' from this modified form, which we will refer to as SRP #11.

Password establishment. A new 'password' is generated with the DH mechanism in C_GenerateKeyPair, leading to a randomly generated private key P and a public key p. Only for reasons of backup might wrapped export of p be permissible, but P can be handled with more relaxation. The two are never used separately however, so there is no real use in treating them any differently.

It is not advised to delete the public key. Although it can be derived in theory from the private key by supplying a 'remote offer' valued 1 to C_DeriveKey, the outcome of this function is not likely to be permitted having the `CKK_DH` key type. TODO: But we don't need that setting, as we will export the public key and use it off-token anyway, since generic mod-exp is not available on the token anyway.

The base value used is the generator g, to establish the relationship $p = g^P$. Both the base value and the modulus are commonly used group parameters for the SRP scheme, with a few standardised in Appendix A of RFC 5054.

Since it is not necessary to export the value of P, it can be created in such a way that PKCS #11 enforces this key concealment. The value p is retained for efficiency purposes, even if it could also be derived on the fly through g^P

with C_DeriveKey.

TODO:EDITED-TILL-HERE

Credentials generation. To derive service access credentials from the password, first generate a random salt s. The value is public, and may be generated on or off the PKCS #11 token. Then, the token's C_DigestXXX functions must be used to compute the hash value $x = H(s)$ either on or off the token; the outcome can be used to calculate $p^{H(s)}$ off-token. The result of that operation is input to C_DeriveKey, where it is raised to the power P in a modular exponentiation operation on the token with C_DigestKey. The outcome is v'', which is supplied in a pair $\langle s, v \rangle$ to the service, together with a username.

Authentication. The client starts off as basic DH with C_GenerateKeyPair. The private key will be known as a and the public key is A. Send A over to the service.

The service generates $B = v'' + g^b$ for a local random secret b and sends B and u to the client, where u is another random scrambling parameter that is publicly shown.

Through the stored values of p and P, the client can now reconstruct v'', which requires the non-extractable secret P; moreover, the outcome cannot be reversed to P due to the properties of the discrete logarithm problem. The value v'' is also known to the service, so it may be exported from the PKCS #11 interface. The client now uses v'' find $B - v''$, which is entirely based on values that the service also sees. The outcome equals g^b, which can be combined with session key a in C_DeriveKey to find g^{ab}, a normal DH result.

The remainder of the computation of S'' consists of using the value $B - v''$ in off-token multiplication and modular exponentiation, all based on the public values $B - v''$, u, $H(s)$ and the locally generated random value a, all of which are safe to do outside of PKCS #11, and there will be two modular exponentiations with secret key P as the exponent, to be conducted on the token behind PKCS #11. It is this last point where the token is a strict requirement in deriving the proper values.

Note that the value S'' is also available on the service, so we consider it public. [**TODO:** How about the $\ldots^P$ and $\ldots^{P^P}$ values, are they 'public' too?] Likewise, the session key $K = H(S)$ is also available to the service and not considered secret.

Note that the value of x'' is absent from the calculations. Instead, the secret key P is used. In a normal calculation the dependency on a secret key might be a liability, but when using PKCS #11 this is not as charged because of its assumed protection of private keys. The value of x'' cannot be derived either, since the protected storage of the password values make it impossible to find the outcome of $H(s)p$ because that is not part of the DH mechanism.

6 Performance Considerations

The use of the more protective private-key encapsulation of PKCS #11 comes at a price of reduced efficiency. Note that this plays particularly at the client side, which means that it is subject to user selection of technology, and more importantly, meaning that scaling problems are not likely to be a problem — as we are talking of an alternative to manual password entry.

Still, a performance indication is useful. The overruling consideration in this sense is how many modular-exponential operations take place. More accurately put, the number of bits that may hold a value in the exponent of these operations. We are going to base our performance considerations on the assumption that the random values s, a, b, u all match the size of the outcome of the hash algorithm H. So, we can count the number of times such bits occur in the exponent of a calculation.

The chief concern is the authentication process. In the original process we find client-side exponents a, x and $a{+}ux$, sized like 1, 1 and 2 hashes, so plain SRP weighs like 4 modular exponentiations with the size of the outcome of H. In SRP #11, the client-side exponents during authentication are $H(s)$, a, a, u, P, P and $H(s)$, which weighs like 7 modular exponentiations.

Clearly, the inability of computing $a + ux$ before the exponention forces in extra computations in return for the added security of not handling x in SRP #11. The computational comlexity of SRP #11 is about 75% higher than for plain SRP. This factor is probably irrelevant in comparison to the (possible) move of the calculation to a piece of hardware. Hardware tends to be either much slower (USB token) or much faster (HSM, crypto co-processor) at performing modular exponentiation than a CPU.

7 Security Considerations

The values that are available as intermediate results in SRP #11 are all 'public', in the sense that the service can also find them based on what it knows. Most computations are based on a generator raised to a (secret or public) exponent, which is considered irreversable on grounds of the infeasability of the discrete logarithm problem. [**TODO:** The values $\ldots^P$ and $\ldots^{P^P}$ might be non-'public'.]

The value v'' is known to the service, and therefore its derivation as $p^{H(s)P}$ during authentication does not reveal information that should be protected by PKCS #11. The value $B - v''$ is 'public', and so is a, so computing $(B - v'')^a$ presents no problems. The same holds for $(B - v'')^u$ and the next step, $((B - v'')^u)^{H(s)}$. The latter value is then twice raised to the value P that is protected behind PKCS #11, but an attacker would need to crack discrete logarithms to reverse this to find P. The outcome of this last operation is equal to the service-side value S multiplied by the inverse of the public-value based $(B - v'')^a$, so having the value once more reveals no new information. The remainder computers K, M_1 and M_2 on both ends, so there is no question about their revelation of new information that ought to be protected PKCS #11.

It is worth noting that a client might turn rogue when it is cracked; in such cases, it might gain the rights to authenticate, but it should not gain access to the passwords through which it can mount future attacks on other locations. This makes the common assumption that the credential protection of PKCS #11 is not cracked; an assumption that can be influenced by a suitable choice of PKCS #11 implementation. Plain SRP and SRP #11 are not different in this respect, and in fact it is common to all DH mechanism. Another point common to all three variations is that thrashing of the random values a and b suffices to thwart future attackers from recovering session keys of traffic preceding the crack on the client (forward secrecy).

The value $p = g^P$ is retained as a 'cache' value. Rogue access to PKCS #11 might consider replacing this value with a more suitable value, to influence future computations based on it. This is not likely to lead to anything useful however; first, it would be noticed when accessing a non-rogue service; second, the value P is still used in computations; and third, the discrete logarithm problem makes it notoriously difficult to derive a value of p that would lead to a desired outcome. A much better attack would be to replace both p and P which is comparable to replacing a password; this however, would not leak any information about the prior value of P and would invalidate access to services setup with it.

Modular exponentiation cannot be reversed, as a result of the discrete logarithm problem. The start of the computation is $B - v''$, which equals g^b and that uses an unknown value b. Raising this to the powers u, $H(s)$, P and P in any order should stir the value even further, but in lieu of the value b on the client and due to the discrete logarithm problem it is unlikely that an attack could be mounted from this angle. **Open Question:** Can we prove that?

In conclusion, the maximum protection that PKCS #11 can deliver has been achieved with SRP #11, under the assumption that what the service knows is sufficiently public. That angle however, is subject of potential crisitism.

It could be argued that the service is in a special position. For one, it generates random string b internally and only publishes the corresponding B; but that alone would not make much difference, an active MITM attack could produce similar values. The vital distinction is that the service knows v', which is assumed to have been relayed to it in a covert manner, and so it is a shared secret between client and service — one that would not be available to anyone on the channel between client and service.

Unfortunately, it is not possible to redesign SRP #11 to treat v'' as a secret value. This can be seen directly in the computations; the construction $(B - v'')^e$, regardless of the exponent e cannot be computed with DH values, because subtraction is not foreseen in this mechanism. This is where PKCS #11 reveals that it was not designed with plain SRP or SRP #11 in mind. And since exponentiation does not distribute through subtraction, neither modular or otherwise, we cannot construct any such mechanism cleverly.

Note that the visibility of v'' is less than perfect, but not truly problematic in itself. The sole reason for treating all data available to the service as public is to avoid putting too much faith in what PKCS #11 can do. Apparently, this is its limitation. Note that this is also caused by something else — the client is the party that originally generated v'' and would be able to do that again.

In conclusion, SRP #11 actually improves the security of SRP by taking the passwords out of sight and placing them behind PKCS #11; in addition, the password-equivalent value x is not available to the SRP #11 computations due to PKCS #11 shielding. The one thing that is not possible, is to also conceal the computation of verifier values v'' from the client, but it has been demonstrated that this would have introduced a false sense of security anyway. In conclusion, SRP #11 establishes precisely the maximum achievable and the maximum desirable protection of SRP credentials.

8 Conclusions

This article introduces SRP #11, a variation on SRP that uses PKCS #11 to protects the password and password-equivalent values that are computed on the client side. It has been shown that a direct implementation of plain SRP with PKCS #11 does protect the original password, but not the password-equivalent value x, which means it is useless. In contrast, the SRP #11 variation achieves precisely the maximum achievable and maximum desirable that PKCS #11 could offer.

It has been shown that the modified generation of salt and verifier combined with the modified client-side authentication computations compute values equivalent to those of plain SRP, and that the service or protocol require no changes. In other words, only the client needs to modify its computations if it intends to use PKCS #11.

In terms of efficiency, the SRP #11 mechanism adds 75% to the computational complexity, relative to plain SRP. When PKCS #11 is implemented on hardware, then it is likely that this choice has an overruling impact on the speed of the computations — they would take longer when a plain USB token is used, and would go down when cryptographic acceleration is built into the hardware.

Open Issue: It remains to be seen if the details of the DH mechanism in PKCS #11 will indeed permit the repeated use of modular exponentiation. An attempt to implement appears to be a useful exercise, in light of the potential achievements of SRP #11; if this mechanism does not work, then chances of finding another that will work are modest, to put it mildly.