Network Working Group W. Ladd
Internet-Draft Cloudflare
Intended status: Informational B. Kaduk, Ed.
Expires: July 2, 2021 Akamai
December 29, 2020

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-16

Abstract

This document describes SPAKE2 which is a protocol for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This method is compatible with any group, is computationally efficient, and SPAKE2 has a security proof. This document predated the CFRG PAKE competition and it was not selected.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 2, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Requirements Notation 2
3. Definition of SPAKE2 2
4. Key Schedule and Key Confirmation 5
5. Per-User M and N 6
6. Ciphersuites 6
7. Security Considerations 9
8. IANA Considerations 9
9. Acknowledgments 9
10. References 9
Appendix A. Algorithm used for Point Generation 11
Appendix B. Test Vectors 13
Authors' Addresses 16

1. Introduction

This document describes SPAKE2, a means for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This password-based key exchange protocol is compatible with any group (requiring only a scheme to map a random input of fixed length per group to a random group element), is computationally efficient, and has a security proof. Predetermined parameters for a selection of commonly used groups are also provided for use by other protocols.

2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Definition of SPAKE2

3.1. Setup

Let G be a group in which the gap Diffie-Hellman (CDH) problem is hard. Suppose G has order p*h where p is a large prime; h will be called the cofactor. Let I be the unit element in G, e.g., the point at infinity if G is an elliptic curve group. We denote the operations in the group additively. We assume there is a representation of elements of G as byte strings: common choices would

be SEC1 [SEC1] uncompressed or compressed for elliptic curve groups or big endian integers of a fixed (per-group) length for prime field DH. We fix two elements M and N in the prime-order subgroup of G as defined in the table in this document for common groups, as well as a generator P of the (large) prime-order subgroup of G. In the case of a composite order group we will work in the quotient group. P is specified in the document defining the group, and so we do not repeat it here.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes, represented as an eight-byte little-endian number. Finally, let nil represent an empty string, i.e., len(nil) = 0.

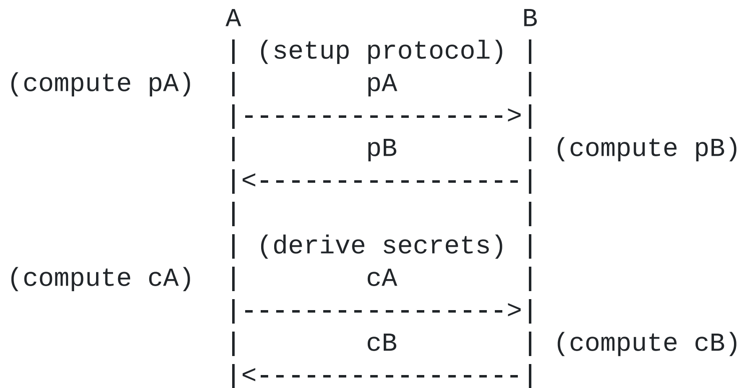
KDF(ikm, salt, info) is a key-derivation function that takes as input a salt, intermediate keying material (IKM), info string, and derived key length L to derive a cryptographic key of length L. MAC is a Message Authentication Code algorithm that takes a secret key and message as input to produce an output. Let Hash be a hash function from arbitrary strings to bit strings of a fixed length. Common choices for H are SHA256 or SHA512 [RFC6234]. Let MHF be a memory-hard hash function designed to slow down brute-force attackers. Scrypt [RFC7914] is a common example of this function. The output length of MHF matches that of Hash. Parameter selection for MHF is out of scope for this document. Section 6 specifies variants of KDF, MAC, and Hash suitable for use with the protocols contained herein.

Let A and B be two parties. A and B may also have digital representations of the parties' identities such as Media Access Control addresses or other names (hostnames, usernames, etc). A and B may share Additional Authenticated Data (AAD) of length at most 2^16 - 1 bits that is separate from their identities which they may want to include in the protocol execution. One example of AAD is a list of supported protocol versions if SPAKE2(+) were used in a higher-level protocol which negotiates use of a particular PAKE. Including this list would ensure that both parties agree upon the same set of supported protocols and therefore prevent downgrade attacks. We also assume A and B share an integer w; typically w = MHF(pw) mod p, for a user-supplied password pw. Standards such as NIST.SP.800-56Ar3 suggest taking mod p of a hash value that is 64 bits longer than that needed to represent p to remove statistical bias introduced by the modulation. Protocols using this specification must define the method used to compute w: it may be necessary to carry out various forms of normalization of the password before hashing [RFC8265]. The hashing algorithm SHOULD be a MHF so as to slow down brute-force attackers.

3.2. Protocol Flow

SPAKE2 is a one round protocol to establish a shared secret with an additional round for key confirmation. Prior to invocation, A and B are provisioned with information such as the input password needed to run the protocol. During the first round, A sends a public share pA to B, and B responds with its own public share pB. Both A and B then derive a shared secret used to produce encryption and authentication keys. The latter are used during the second round for key confirmation. (Section 4 details the key derivation and confirmation steps.) In particular, A sends a key confirmation message cA to B, and B responds with its own key confirmation message cB. Both parties MUST NOT consider the protocol complete prior to receipt and validation of these key confirmation messages.

This sample trace is shown below.



3.3. SPAKE2

To begin, A picks x randomly and uniformly from the integers in [0,p), and calculates X=x*P and S=w*M+X, then transmits pA=S to B.

B selects y randomly and uniformly from the integers in [0,p), and calculates Y=y*P, T=w*N+Y, then transmits pB=T to A.

Both A and B calculate a group element K. A calculates it as h*x*(T-w*N), while B calculates it as h*y*(S-w*M). A knows S because it has received it, and likewise B knows T. The multiplication by h prevents small subgroup confinement attacks by computing a unique value in the quotient group. This is a common mitigation against this kind of attack.

K is a shared value, though it MUST NOT be used as a shared secret. Both A and B must derive two shared secrets from the protocol transcript. This prevents man-in-the-middle attackers from inserting

themselves into the exchange. The transcript TT is encoded as follows:

```
        TT = len(A) || A
          || len(B) || B
          || len(S) || S
          || len(T) || T
          || len(K) || K
          || len(w) || w
```

Here w is encoded as a big endian number padded to the length of p. This representation prevents timing attacks that otherwise would reveal the length of w. len(w) is thus a constant. We include it for consistency.

If an identity is absent, it is encoded as a zero-length string. This MUST only be done for applications in which identities are implicit. Otherwise, the protocol risks Unknown Key Share attacks (discussion of Unknown Key Share attacks in a specific protocol is given in [I-D.ietf-mmusic-sdp-uks]).

Upon completion of this protocol, A and B compute shared secrets Ke, KcA, and KcB as specified in Section 4. A MUST send B a key confirmation message so both parties agree upon these shared secrets. This confirmation message F is computed as a MAC over the protocol transcript TT using KcA, as follows: F = MAC(KcA, TT). Similarly, B MUST send A a confirmation message using a MAC computed equivalently except with the use of KcB. Key confirmation verification requires computing F and checking for equality against that which was received.

4. Key Schedule and Key Confirmation

The protocol transcript TT, as defined in Section Section 3.3, is unique and secret to A and B. Both parties use TT to derive shared symmetric secrets Ke and Ka as Ke || Ka = Hash(TT), with |Ke| = |Ka|. The length of each key is equal to half of the digest output, e.g., 128 bits for SHA-256.

Both endpoints use Ka to derive subsequent MAC keys for key confirmation messages. Specifically, let KcA and KcB be the MAC keys used by A and B, respectively. A and B compute them as KcA || KcB = KDF(Ka,nil, "ConfirmationKeys" || AAD), where AAD is the associated data each given to each endpoint, or nil if none was provided. The length of each of KcA and KcB is equal to half of the underlying hash output length, e.g., |KcA| = |KcB| = 128 bits for HKDF(SHA256).

The resulting key schedule for this protocol, given transcript TT and additional associated data AAD, is as follows.

```
TT  -> Hash(TT) = Ke || Ka
AAD -> KDF(nil, Ka, "ConfirmationKeys" || AAD) = KcA || KcB
```

A and B output Ke as the shared secret from the protocol. Ka and its derived keys are not used for anything except key confirmation.

5. Per-User M and N

To avoid concerns that an attacker needs to solve a single ECDH instance to break the authentication of SPAKE2, a variant based on using [I-D.irtf-cfrg-hash-to-curve] is also presented. In this variant, M and N are computed as follows.

```
M = h2c(Hash("M for SPAKE2" || len(A) || A || len(B) || B))
N = h2c(Hash("N for SPAKE2" || len(A) || A || len(B) || B))
```

In addition M and N may be equal to have a symmetric variant. The security of these variants is examined in [MNVAR]. This variant may not be suitable for protocols that require the messages to be exchanged symmetrically and do not know the exact identity of the parties before the flow begins.

6. Ciphersuites

This section documents SPAKE2 ciphersuite configurations. A ciphersuite indicates a group, cryptographic hash algorithm, and pair of KDF and MAC functions, e.g., SPAKE2-P256-SHA256-HKDF-HMAC. This ciphersuite indicates a SPAKE2 protocol instance over P-256 that uses SHA256 along with HKDF [RFC5869] and HMAC [RFC2104] for G, Hash, KDF, and MAC functions, respectively.

G	Hash	KDF	MAC
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-384	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-384	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-512	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
edwards25519 [RFC7748]	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
edwards448 [RFC7748]	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]

Table 1: SPAKE2 Ciphersuites

The following points represent permissible point generation seeds for the groups listed in the Table Table 1, using the algorithm presented in Appendix A. These bytestrings are compressed points as in [SEC1] for curves from [SEC1].

For P256:

```
M =
02886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f
seed: 1.2.840.10045.3.1.7 point generation seed (M)

N =
03d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b49
seed: 1.2.840.10045.3.1.7 point generation seed (N)
```

```
For P384:

M =
030ff0895ae5ebf6187080a82d82b42e2765e3b2f8749c7e05eba366434b363d3dc
36f15314739074d2eb8613fceec2853
seed: 1.3.132.0.34 point generation seed (M)

N =
02c72cf2e390853a1c1c4ad816a62fd15824f56078918f43f922ca21518f9c543bb
252c5490214cf9aa3f0baab4b665c10
seed: 1.3.132.0.34 point generation seed (N)

For P521:

M =
02003f06f38131b2ba2600791e82488e8d20ab889af753a41806c5db18d37d85608
cfae06b82e4a72cd744c719193562a653ea1f119eef9356907edc9b56979962d7aa
seed: 1.3.132.0.35 point generation seed (M)

N =
0200c7924b9ec017f3094562894336a53c50167ba8c5963876880542bc669e494b25
32d76c5b53dfb349fdf69154b9e0048c58a42e8ed04cef052a3bc349d95575cd25
seed: 1.3.132.0.35 point generation seed (N)

For edwards25519:

M =
d048032c6ea0b6d697ddc2e86bda85a33adac920f1bf18e1b0c6d166a5cecdaf
seed: edwards25519 point generation seed (M)

N =
d3bfb518f44f3430f29d0c92af503865a1ed3281dc69b35dd868ba85f886c4ab
seed: edwards25519 point generation seed (N)

For edwards448:

M =
b6221038a775ecd007a4e4dde39fd76ae91d3cf0cc92be8f0c2fa6d6b66f9a12
942f5a92646109152292464f3e63d354701c7848d9fc3b8880
seed: edwards448 point generation seed (M)

N =
6034c65b66e4cd7a49b0edec3e3c9ccc4588afd8cf324e29f0a84a072531c4db
f97ff9af195ed714a689251f08f8e06e2d1f24a0ffc0146600
seed: edwards448 point generation seed (N)
```

7. Security Considerations

A security proof of SPAKE2 for prime order groups is found in [REF], reducing the security of SPAKE2 to the gap Diffie-Hellman assumption. Note that the choice of M and N is critical for the security proof. The generation methods specified in this document are designed to eliminate concerns related to knowing discrete logs of M and N.

Elements received from a peer MUST be checked for group membership: failure to properly validate group elements can lead to attacks. It is essential that endpoints verify received points are members of G.

The choices of random numbers MUST BE uniform. Randomly generated values (e.g., x and y) MUST NOT be reused; such reuse may permit dictionary attacks on the password. To generate these uniform numbers rejection sampling is recommended. Some implementations of elliptic curve multiplication may leak information about the length of the scalar: these MUST NOT be used.

SPAKE2 does not support augmentation. As a result, the server has to store a password equivalent. This is considered a significant drawback in some use cases.

The HMAC keys in this document are shorter than recommended in [RFC8032]. This is appropriate as the difficulty of the discrete logarithm problem is comparable with the difficulty of brute forcing the keys.

8. IANA Considerations

No IANA action is required.

9. Acknowledgments

Special thanks to Nathaniel McCallum and Greg Hudson for generation of M and N, and Cris Wood for test vectors. Thanks to Mike Hamburg for advice on how to deal with cofactors. Greg Hudson also suggested the addition of warnings on the reuse of x and y. Thanks to Fedor Brunner, Adam Langley,Liliya Akhmetzyanova, and the members of the CFRG for comments and advice. Chris Wood contributed substantial text and reformatting to address the excellent review comments from Kenny Paterson.

10. References

10.1. Normative References

[I-D.irtf-cfrg-hash-to-curve]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-05 (work in progress), November 2019.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/info/rfc4493>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[I-D.ietf-mmusic-sdp-uks]
Thomson, M. and E. Rescorla, "Unknown Key Share Attacks on uses of TLS with the Session Description Protocol (SDP)", draft-ietf-mmusic-sdp-uks-07 (work in progress), August 2019.

[MNVAR] Abdalla, M., Barbosa, M., Bradley, T., Jarecki, S., Katz, J., and J. Xu, "Universally Composable Relaxed Password Authentication", August 2020.

Appears in Micciancio D., Ristenpart T. (eds) Advances in Cryptology -CRYPTO 20202. Crypto 20202. Lecture notes in Computer Science volume 12170. Springer.

[REF] Abdalla, M. and D. Pointcheval, "Simple Password-Based Encrypted Key Exchange Protocols.", Feb 2005.

Appears in A. Menezes, editor. Topics in Cryptography-CT-RSA 2005, Volume 3376 of Lecture Notes in Computer Science, pages 191-208, San Francisco, CA, US. Springer-Verlag, Berlin, Germany.

[RFC8265] Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", RFC 8265, DOI 10.17487/RFC8265, October 2017, <https://www.rfc-editor.org/info/rfc8265>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.

[TDH] Cash, D., Kiltz, E., and V. Shoup, "The Twin-Diffie Hellman Problem and Applications", 2008.

EUROCRYPT 2008. Volume 4965 of Lecture notes in Computer Science, pages 127-145. Springer-Verlag, Berlin, Germany.

Appendix A. Algorithm used for Point Generation

This section describes the algorithm that was used to generate the points (M) and (N) in the table in Section 6.

For each curve in the table below, we construct a string using the curve OID from [RFC5480] (as an ASCII string) or its name, combined with the needed constant, for instance "1.3.132.0.35 point generation seed (M)" for P-512. This string is turned into a series of blocks by hashing with SHA256, and hashing that output again to generate the next 32 bytes, and so on. This pattern is repeated for each group and value, with the string modified appropriately.

A byte string of length equal to that of an encoded group element is constructed by concatenating as many blocks as are required, starting from the first block, and truncating to the desired length. The byte string is then formatted as required for the group. In the case of Weierstrass curves, we take the desired length as the length for representing a compressed point (section 2.3.4 of [SEC1]), and use the low-order bit of the first byte as the sign bit. In order to obtain the correct format, the value of the first byte is set to 0x02 or 0x03 (clearing the first six bits and setting the seventh bit), leaving the sign bit as it was in the byte string constructed by concatenating hash blocks. For the [RFC8032] curves a different procedure is used. For edwards448 the 57-byte input has the least-significant 7 bits of the last byte set to zero, and for edwards25519 the 32-byte input is not modified. For both the [RFC8032] curves the (modified) input is then interpreted as the representation of the group element. If this interpretation yields a valid group element with the correct order (p), the (modified) byte string is the output. Otherwise, the initial hash block is discarded and a new byte string constructed from the remaining hash blocks. The procedure of constructing a byte string of the appropriate length, formatting it as required for the curve, and checking if it is a valid point of the correct order, is repeated until a valid element is found.

The following python snippet generates the above points, assuming an elliptic curve implementation following the interface of Edwards25519Point.stdbase() and Edwards448Point.stdbase() in Appendix A of [RFC8032]:

```
def iterated_hash(seed, n):
    h = seed
    for i in range(n):
        h = hashlib.sha256(h).digest()
    return h

def bighash(seed, start, sz):
    n = -(-sz // 32)
    hashes = [iterated_hash(seed, i) for i in range(start, start + n)]
    return b''.join(hashes)[:sz]

def canon_pointstr(ecname, s):
    if ecname == 'edwards25519':
        return s
    elif ecname == 'edwards448':
        return s[:-1] + bytes([s[-1] & 0x80])
    else:
        return bytes([(s[0] & 1) | 2]) + s[1:]

def gen_point(seed, ecname, ec):
    for i in range(1, 1000):
        hval = bighash(seed, i, len(ec.encode()))
        pointstr = canon_pointstr(ecname, hval)
        try:
            p = ec.decode(pointstr)
            if p != ec.zero_elem() and p * p.l() == ec.zero_elem():
                return pointstr, i
        except Exception:
            pass
```

Appendix B. Test Vectors

This section contains test vectors for SPAKE2 using the P256-SHA256-HKDF-HMAC ciphersuite. (Choice of MHF is omitted and values for w,x and y are provided directly.) All points are encoded using the uncompressed format, i.e., with a 0x04 octet prefix, specified in [SEC1] A and B identity strings are provided in the protocol invocation.

B.1. SPAKE2 Test Vectors

```
spake2: A='server', B='client'
w = 0x19eed1f4855a0b7e22096a04936c217a5f0cfe480ae626b9d4427dce9373b3f3
x = 0x79bfb7cd97b3c592698af4e8aa2ed20e9f3873cf33310b2b6a9b3f0694b54fd1
S = 0x0498156fb8a640f7b4d656d5c38e1f69fc9db9aefa2537a92462172ed4dc197f
ea356b628fcbc93df133b6c54317e0e805eaa71cb1a23cc2ffc287247c836855ab
y = 0x199f69b150e0aa7f43d41ecba48ce2242aaa462cb106533845a1e9015fee38ce
T = 0x040ccd1b742844109eafa973972bef13844124e56163c225e529ec776ebaf1fb
```

```
1142e1dc4d792c1762998290e45a8419a8059aa45004d9ae099dada77736bcd65f
K = 0x04e896fa87681d37fe9c3e68e9fa406265e63dd0b1b812c802b0bba8557e5bcf
b90d7ca84d3d09eea0fe84ff6e12b161f282a0393c2f94d5b6a6230e115e0e7ce0
TT = 0x06000000000000007365727665720600000000000000636c69656e744100000
0000000000498156fb8a640f7b4d656d5c38e1f69fc9db9aefa2537a92462172ed4dc1
97fea356b628fcbc93df133b6c54317e0e805eaa71cb1a23cc2ffc287247c836855ab4
100000000000000040ccd1b742844109eafa973972bef13844124e56163c225e529ec7
76ebaf1fb1142e1dc4d792c1762998290e45a8419a8059aa45004d9ae099dada77736b
cd65f41000000000000000004e896fa87681d37fe9c3e68e9fa406265e63dd0b1b812c80
2b0bba8557e5bcfb90d7ca84d3d09eea0fe84ff6e12b161f282a0393c2f94d5b6a6230
e115e0e7ce0200000000000000019eed1f4855a0b7e22096a04936c217a5f0cfe480ae
626b9d4427dce9373b3f3
Hash(TT)=c5a9dc83de36d046a9387274344ba6b9d9ff320226b3b698f27d67c5dd563459
Ke = 0xc5a9dc83de36d046a9387274344ba6b9
Ka = 0xd9ff320226b3b698f27d67c5dd56
KcA = 0x94902da13b202b647bd97486653e2145
KcB = 0x404433f5a0a01ce4a8a8b42a41a8a853
A conf = 0xd4bf4f3a13416096b2be325bbe5e31fe277c1733078beb768a830d2df5abf0b5
B conf = 0x2e4841b5353d1c27625c6b94bda481a8f453bcceb51ca6455d661675a751ed0d

spake2: A='', B='client'
w = 0x1af09ee09d36e14781d6af24e17eb927141148dab79d749f6a15a37cbcaebb49
x = 0x02877cda92b90888c081feb5d84fd278820bcce3914f8bb58af03e324aefeb1e
S = 0x04350422b3f16b4a030defd0a9b689bb2454a2a24974889583d9c47653ac5bbe
f5a0d33c8284aec0d4906d8ea22de211d4a60c8e0d6dd3c4d21114a059a7e4c753
y = 0x5d4fc1ded262f19b33c2790378392d43e1967dab8db4a5c8459262eee0635a35
T = 0x04321f59e8ae418a913005a860779a1e2c567715325a91ec75f6625a6dca7a7b
25ddb61333c6f42c9ade343dfdc21cfc88c97edf7a56c2d9d2e309d33542e8f04d
K = 0x04105ff327fcdb0bdd576f894bc2789b88b39ea6b24fd06062defeb7de369ddf
8555d1e957ef2e314780edc92ff8827f89248a16941265f21752cd9330526b86b7
TT = 0x000000000000000006000000000000000636c69656e74410000000000000000043
50422b3f16b4a030defd0a9b689bb2454a2a24974889583d9c47653ac5bbef5a0d33c8
284aec0d4906d8ea22de211d4a60c8e0d6dd3c4d21114a059a7e4c7534100000000000
00004321f59e8ae418a913005a860779a1e2c567715325a91ec75f6625a6dca7a7b25d
db61333c6f42c9ade343dfdc21cfc88c97edf7a56c2d9d2e309d33542e8f04d4100000
00000000004105ff327fcdb0bdd576f894bc2789b88b39ea6b24fd06062defeb7de369
ddf8555d1e957ef2e314780edc92ff8827f89248a16941265f21752cd9330526b86b72
0000000000000001af09ee09d36e14781d6af24e17eb927141148dab79d749f6a15a37
cbcaebb49
Hash(TT)=22d594d6cd6016646fe76a4a2f0a908b90f83701e4962aed6095abd1c72a87ac
Ka = 0x90f83701e4962aed6095abd1c72a
KcA = 0xdaa3b64d68d1aeaa7ca138bdf4df44d1
KcB = 0x13606e2849fc3aedb009da164af6650e
A conf = 0xd5dff21e4d2df66b1ccf0589aece62f8dc4100e0113ecef666083b4be8aa521a
B conf = 0x6bb1db855aaf7bb565255cd4d830867d3db1fd6b75147953c59e5b8d94aab96a


spake2: A='server', B=''
```

```
w = 0x2a7ae95677292de1b1c3e073d4f446cafc49686a1ac15be4c4a7f7ff68be7eb4
x = 0xa152ba5343eb60e3f0867cdd798f4ccf6c97107ec092b9029210f94e082d009e
S = 0x04d646aa145fee782fb65115b98265833503bd3acd8ce825f9655c51f89cd7f1
83935be0c56300e27522411211814085d2e72ffaa2b7dd8b3fe8bd2a679505c538
y = 0xe6b65dec48caf62859c5c004822dc9322c0c1457e2a1c2ddf35db83bc4082c00
T = 0x04e6c8df6777bcf56a7a5a1dd25a9b2aafeb7bd04460c7a6c27d030f021c146d
a575116155217d99157398c9a281d459d5d5742767ff079e1f7b1466f83afb8f8f
K = 0x043e6f809c51415045d96135997c3b2b8aa203152134b24351dcc34638e3998a
9313d63aa398730bda790bd9494d51aa5cfc7a2a504d87b553d639894d2e485dbe
TT = 0x06000000000000007365727665720000000000000000410000000000000004d
646aa145fee782fb65115b98265833503bd3acd8ce825f9655c51f89cd7f183935be0c
56300e27522411211814085d2e72ffaa2b7dd8b3fe8bd2a679505c5384100000000000
00004e6c8df6777bcf56a7a5a1dd25a9b2aafeb7bd04460c7a6c27d030f021c146da57
5116155217d99157398c9a281d459d5d5742767ff079e1f7b1466f83afb8f8f4100000
000000000043e6f809c51415045d96135997c3b2b8aa203152134b24351dcc34638e39
98a9313d63aa398730bda790bd9494d51aa5cfc7a2a504d87b553d639894d2e485dbe2
0000000000000002a7ae95677292de1b1c3e073d4f446cafc49686a1ac15be4c4a7f7f
f68be7eb4
Hash(TT)=178508e1cec7e951bfa06584c2912e5ca722d894e0c0a00ef36d1531f1bdcd6a
Ka = 0xa722d894e0c0a00ef36d1531f1bd
KcA = 0x92e30a69434569c7f62bc33241458abd
KcB = 0x4aae21b4ba3afc536b711730663f7be2
A conf = 0x94cc7422f19c21110a272e5ba28218d672779dd8f84c758346364611eaa5a595
B conf = 0xc3ef73a0073b67347678da7eacbf34bfbb15e33e7277b35f915ece0c8e6a8382

spake2: A='', B=''
w = 0x94b84fe32e2a40b3cacaaf0654f315f4b59b327fe7a5f2377e4c8eeaf704bb22
x = 0x69fe68d9d2801bffbe0d39cf176343eb4926b33fcaec9878dae8c50fa30cf657
S = 0x04842fb511920771b8bb5598cf86c039c656d96bf17fcc0ce782a8766d2c3809
b6cca257d6892273dd9598b2b02cc807a82a23f57adf20fd86cffc2de5a6b424af
y = 0x7865d01ef38cc20c7032f6843d6cb137b710c947fe1295e373ce7166a0f3abd7
T = 0x04dfbe6ee311032dd0afcaa64dc9c2f0c0f0731faaa347f41d9ab9473ad57028
bd6adb4276e893971fe9ed07eddf9ee2fd9b5ba50b4ff38832832b05f054acddc8
K = 0x04ff5129244237f0b2d9f365bfee3d5af1d39eee85cfbe50b6f03fd2c6fc5fef
4d039a2c29e686f2d0707fb29c88986f0d1e31f8b320f723fe2fef4e5681f20370
TT = 0x00000000000000000000000000000000410000000000000004842fb51192077
1b8bb5598cf86c039c656d96bf17fcc0ce782a8766d2c3809b6cca257d6892273dd959
8b2b02cc807a82a23f57adf20fd86cffc2de5a6b424af410000000000000004dfbe6ee
311032dd0afcaa64dc9c2f0c0f0731faaa347f41d9ab9473ad57028bd6adb4276e8939
71fe9ed07eddf9ee2fd9b5ba50b4ff38832832b05f054acddc8410000000000000004f
f5129244237f0b2d9f365bfee3d5af1d39eee85cfbe50b6f03fd2c6fc5fef4d039a2c2
9e686f2d0707fb29c88986f0d1e31f8b320f723fe2fef4e5681f203702000000000000
00094b84fe32e2a40b3cacaaf0654f315f4b59b327fe7a5f2377e4c8eeaf704bb22
Hash(TT)= 3d9463ed0efada5fdab08d5c99bc80770b4098396c542ce89c0d2a7a08b83ae5
Ka = 0x0b4098396c542ce89c0d2a7a08b8
KcA = 0xa03095d310e975f0e2301a41f1b0e358
KcB = 0x4576319b6aacfdbf2ead7ce42f63f513
A conf = 0x5dc0efae13f0d5418c3bf56dd04ea15d9bc0a3aa86e85119dbf7cdc63609c6ee
```

B conf = 0x370bf83e684366223ed33d43dd8b9b605cf8a309c6ab97af8cb2451d94aa08fb

Authors' Addresses

Watson Ladd
Cloudflare

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu