Network Working Group W. Ladd
Internet-Draft Cloudflare
Intended status: Informational B. Kaduk, Ed.
Expires: March 13, 2021 Akamai
September 9, 2020

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-13

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

Copyright Notice

Table of Contents

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 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.

```
                A                   B
                | (setup protocol) |
  (compute pA)  |        pA        |
                |----------------->|
                |        pB        | (compute pB)
                |<-----------------|
                |                  |
                | (derive secrets) |
  (compute cA)  |        cA        |
                |----------------->|
                |        cB        | (compute cB)
                |<-----------------|
```

3.3. SPAKE2

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

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

Both A and B calculate a group element K. A calculates it as h*x*(S-w*N), while B calculates it as h*y*(T-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
```

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(nil, Ka, "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 KDF output, 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) = Ka || Ke
    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].

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 is 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.

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

8. IANA Considerations

No IANA action is required.

9. Acknowledgments

Special thanks to Nathaniel McCallum and Greg Hudson for generation of 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, 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. and M. Barbosa, "Perfect Forward Security of SPAKE2", Oct 2019.

IACR eprint 2019/1194

[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 and w0,w1 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='client', B='server')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x04ac6827b3a9110d1e663bcd4f5de668da34a9f45e464e99067bbea53f1ed4
d8abbdd234c05b3a3a8a778ee47f244cca1a79acb7052d5e58530311a9af077ba179
T = 0x04e02acfbbfb081fc38b5bab999b5e25a5ffd0b1ac48eae24fcc8e49ac5e0d
8a790914419a100e205605f9862daa848e99cea455263f0c6e06bc5a911f3e10a16b
```

```
Y = 0x0413c45ab093a75c4b2a6e71f957eec3859807858325258b0fa43df5a6efd2
63c59b9c1fbfd55bc5e75fd3e7ba8af6799a99b225fe6c30e6c2a2e0ab4962136ba8
S = 0x047aad50ba7bd6a5eacbead7689f7146f1a4219fa071cce1755f80280cc6c3
a5a73cf469f2a294a0b74a5c07054585ccd447f3f633d8631f3bf43442449e9efeba
TT = 0x0600000000000000636c69656e7406000000000000007365727665724100
00000000000047aad50ba7bd6a5eacbead7689f7146f1a4219fa071cce1755f80280
cc6c3a5a73cf469f2a294a0b74a5c07054585ccd447f3f633d8631f3bf43442449e9
efeba41000000000000004e02acfbbfb081fc38b5bab999b5e25a5ffd0b1ac48eae
24fcc8e49ac5e0d8a790914419a100e205605f9862daa848e99cea455263f0c6e06b
c5a911f3e10a16b41000000000000004d01fc08bbae9b6abe2f4d6893cc9f810433
2e19be5f5881c6b9f077e1feff55023da74db65fae320fad8f0dd38e1323f5336f3f
53c9c9dec06710f18f556bd2020000000000000007741cf8c80b9bee583abac3d38d
aa6b807fed38b06580cb75ee85319d25fede6
Ka = 0x2b5e350c58d530c3586f75bf2a155c4b
Ke = 0x238509f7adf0dc72500b2d1315737a27
KcA = 0xc33d2ef8e37a7e545c14c7fcfdc9db94
KcB = 0x18a81cec7eb83416db6615cb3bc03fcb
MAC(A) = 0x29e9a63d243f2f0db5532d2eb0dbaa617803b85feb31566d0cb9457e3
03bcfa6
MAC(B) = 0x487e4cbe98b6287272d043e169a19b6c4682d0481c92f53f1ee03d4b8
6c3f43e

SPAKE2(A='client', B='')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x048b5d7b44b02c4c868f4486ec55bd2380ec34cd5fa5dbff1079a79097e305
0b34fa91272331729357c86cbb30d371e252dc915aeaa314921b1f09f74816f96a12
T = 0x04839f44931b88d12769e601d0ec480b6c9ea95e70ba361ba14bf513e5186a
6c302e6f409bd01f1030ad3cdac1e08965217e430ca7f9bce698111ae8a4d0530efd
Y = 0x0446419d63037d0bbaca224f89987c776bfea2e0913ccda0790079212f476d
6fd1ec997a02821a804f885e4f29b172b27c92251d883efe201cae106c239108c0c7
S = 0x042926b2dbcc5d0cb23ca123cc4133242f2998439af5380434a4bd5fd76fbb
c030b5563218d0184fa3fd303482a679c9555ccea41098b26b6ee16fe35c792b1fda
TT = 0x0600000000000000636c69656e744100000000000000042926b2dbcc5d0cb
23ca123cc4133242f2998439af5380434a4bd5fd76fbbc030b5563218d0184fa3fd3
03482a679c9555ccea41098b26b6ee16fe35c792b1fda41000000000000004839f4
4931b88d12769e601d0ec480b6c9ea95e70ba361ba14bf513e5186a6c302e6f409bd
01f1030ad3cdac1e08965217e430ca7f9bce698111ae8a4d0530efd4100000000000
000041d9e3c88db68247ab50264a6090e2e524bda3049dbc53c4df708e37bd76913b
8cf5954c4d0f835331f185fef4ff1c6115cf0eb8ce27e8224bf5f76c75b182308200
00000000000007741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee8531
9d25fede6
Ka = 0xfc8482d5d7623a75ad09721d631d1392
Ke = 0x93f618fe24d0d5a54b320f498dbd3ecb
KcA = 0x75b20fc4205d6217a22156f918dd03b1
KcB = 0x3bf3a5d3876d9d12dc54cab927acd5f7
MAC(A) = 0xd4994b751eb832b2836ad674cd615c643053278864a63e263bc2f324b
9a04ddd
```

```
MAC(B) = 0x23cf761999b7603adf5507b50c9bda4eaabe8fa7a9ad0280729dfcd00
8b2bf05

SPAKE2(A='', B='server')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x0465e8b4709ba622bc97af5dde3b41757c2114bfc5abb10141245cb01d62ca
0d7360e1169cd518f9351bbfa44a66cc5f3bcb60661a04f39b04a3d504046db67884
T = 0x0482f64286419ff46362faf781776edf908740b8ff612e0bfe3c90cdc553ba
db7f882a4110ee71fa13a693b5ce96ceba5798636555d074648d4521e3b63dc14872
Y = 0x041aa11299692627a7cac122d4c14606ff700a8be6a0fb1c42f3762d629893
ab3ca51e4a48c798fc8c6b9dcfda1ad33099ed2f73abe6b3500ce383f54011430c26
S = 0x04adba3c3b9a74d9769651d09aedb37d22b9471b9e408e2b98fdd4188c12fa
c731e9dc87e029f7dee0213660ddf0791f50dd8fd32f7152015be0489125b3831b4b
TT = 0x06000000000000007365727665724100000000000000004adba3c3b9a74d97
69651d09aedb37d22b9471b9e408e2b98fdd4188c12fac731e9dc87e029f7dee0213
660ddf0791f50dd8fd32f7152015be0489125b3831b4b41000000000000000482f64
286419ff46362faf781776edf908740b8ff612e0bfe3c90cdc553badb7f882a4110e
e71fa13a693b5ce96ceba5798636555d074648d4521e3b63dc148724100000000000
00004a406929024a5275372531c85c54fd222f35bfdb1cdf1bd1abe82d5c837744d9
3ea2979962eb374d4feda37b178e91711c52edd453178cf69748e0a3d9ef073c2200
00000000000007741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee8531
9d25fede6
Ka = 0xcd9c33c6329761919486d0041faccb56
Ke = 0xa08125eeed51c61ad93b2ff7d8ec3cd5
KcA = 0x60056386cbe06ba199fa6aef81dfb273
KcB = 0x5e5a591b4426d47190aecb2fc4527140
MAC(A) = 0xf0dcfb4fa874e3fcbadd44b6eb26a64d1d5c6e50034934934551f172d
3cdc50e
MAC(B) = 0x52e7a505c0b73db656108554a854c3f33bfb01edcc1ee52aa27ceb1cb
ef7f47b

SPAKE2(A='', B='')
w = 0x7741cf8c80b9bee583abac3d38daa6b807fed38b06580cb75ee85319d25fed
e6
X = 0x04fbeb44d6b772fa390fcced51be7316107e608ddf4ab5dcc9f1b2e24bf667
7f3232cdeeb39a61621a9e48028997d449894212eb54b6f12bdbd9baf8f1c909a740
T = 0x04887af8439d743215f26d48314835b024b9301ea508eac3a339241672fbba
09f63e155b1df5d31ccc63babafc00ffff6e258c692aed84a859fd4960d99fcec777
Y = 0x04bb4727c5c5c50ae34d5148ec6797e5ebf93ae51c5c6cfd48579c41436823
1ac8769142bf6a0109bd2b86dd901c6054629ce2c6b982326c9cd9a3685c4cf0640d
S = 0x04665b5101132528be32f4b4762d6ae80273bbe74e151fc2320da373e146ee
cd33038ff8099782f3781160244672cb43b4d9f2007da9b617c1890845440da0ca53
TT = 0x410000000000000004665b5101132528be32f4b4762d6ae80273bbe74e151
fc2320da373e146eecd33038ff8099782f3781160244672cb43b4d9f2007da9b617c
1890845440da0ca5341000000000000000004887af8439d743215f26d48314835b024b
9301ea508eac3a339241672fbba09f63e155b1df5d31ccc63babafc00ffff6e258c6
92aed84a859fd4960d99fcec77741000000000000000004aacd2378990cecd338c7cac
```

```
   d132ce633bc424ac5d4ab32f539ccf31f15deef62463253790e139b461c5137944fc
   6a5ffd895dbe0d3960b01f6d662fc41057a70200000000000000007741cf8c80b9bee
   583abac3d38daa6b807fed38b06580cb75ee85319d25fede6
   Ka = 0x16b10f1541c24c630f462f7e0aa57ddf
   Ke = 0xb7ae8b61938e3dfad8b9ce1d2865533f
   KcA = 0x3398d6c7de402a9ae89a4594d5576c21
   KcB = 0x6894ab44d7ba7f3a40a772d1476593d9
   MAC(A) = 0x12fce7f0aecc1dba393a7e5612e6357becc5e3d07cd41ffd35c6d652f
   29cde60
   MAC(B) = 0xac36c6d186c3b824f4cfe099f035cf3aed4162d08886d32fa1806e5bf
   4015255
```

Authors' Addresses

Watson Ladd
Cloudflare

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu