Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

W. Ladd
UC Berkeley
B. Kaduk, Ed.
Akamai

March 11, 2019

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-08

Abstract

This document describes SPAKE2 and its augmented variant SPAKE2+, which are protocols 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 prime order group, is computationally efficient, and SPAKE2 (but not SPAKE2+) has a security proof.

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 September 12, 2019.

Copyright Notice

Copyright (c) 2019 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

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 computational 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 compressed [SEC1] 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. 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 5 specifies variants of KDF, MAC, Hash, and MHF 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 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.

We present two protocols below. Note that it is insecure to use the same password with both protocols; passwords MUST NOT be used for both SPAKE2 and SPAKE2+.

3.2. 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 T to B. Upon receipt of T, B computes T*h and aborts if the result is equal to I. (This ensures T is in the prime order subgroup of G.)

B selects y randomly and uniformly from the integers in [0,p), and calculates Y=y*P, S=w*N+Y, then transmits S to A. Upon receipt of S, A computes S*h and aborts if the result is equal to I.

Both A and B calculate a group element K. A calculates it as x*(S-wN), while B calculates it as y*(T-w*M). A knows S because it has received it, and likewise B knows T. A and B multiply protocol messages from each peer by h so as to avoid small subgroup attacks, but the result of the multiplication is not used for operations other than the comparison against I and the non-multiplied value is used in subsequent calculations.

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 K and 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 omitted from the transcript entirely.

For example, if both A and B are absent, then TT = len(S) || S || len(T) || T || len(K) || K || len(w) || w. Likewise, if only A is absent, TT = len(B) || B || len(S) || S || len(T) || T || len(K) || K || len(w) || w. 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 protocl 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.

3.3. SPAKE2+

This protocol appears in [TDH]. We use the same setup as for SPAKE2, except that we have two secrets, w0 and w1, derived by hashing the password pw with the identities of the two participants, A and B. Specifically, w0s || w1s = MHF(len(pw) || pw || len(A) || A || len(B) || B), and then computing w0 = w0s mod p and w1 = w1s mod p. The length of each of w0s and w1s is equal to half of the MHF output, e.g., |w0s| = |w1s| = 128 bits for scrypt. w0 and w1 MUST NOT equal I. If they are, they MUST be iteratively regenerated by computing w0s || w1s = MHF(len(pw) || pw || len(A) || A || len(B) || B || 0x0000), where 0x0000 is 16-bit increasing counter. This process must repeat until valid w0 and w1 are produced. B stores L=w1*P and w0.

When executing SPAKE2+, A selects x uniformly at random from the numbers in the range [0, p), and lets X=x*P+w0*M, then transmits X to B. Upon receipt of X, A computes h*X and aborts if the result is equal to I. B then selects y uniformly at random from the numbers in [0, p), then computes Y=y*P+w0*N, and transmits Y to A. Upon receipt of Y, A computes Y*h and aborts if the result is equal to I.

A computes Z as x*(Y-w0*N), and V as w1*(Y-w0*N). B computes Z as y*(X- w0*M) and V as y*L. Both share Z and V as common keys. It is essential that both Z and V be used in combination with the transcript to derive the keying material. The protocol transcript encoding is shown below.

```
TT = len(A) || A || len(B) || B || len(X) || X || len(Y) || Y
 || len(Z) || Z || len(V) || V || len(w0) || w0
```

As in Section 3.2, inclusion of A and B in the transcript is optional depending on whether or not the identities are implicit.

Upon completion of this protocol, A and B follow the same key derivation and confirmation steps as outlined in Section 3.2.

4. Key Schedule and Key Confirmation

The protocol transcript TT, as defined in Sections Section 3.3 and Section 3.2, 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). The length of each key is equal to half of the digest output, e.g., |Ke| = |Ka| = 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) = 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. Ciphersuites

This section documents SPAKE2 and 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	MHF
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]

P-384	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-384	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-512	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
edwards25519 [RFC7748]	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
edwards448 [RFC7748]	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]	scrypt [RFC7914]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]	scrypt [RFC7914]

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)
```

6. Security Considerations

A security proof of SPAKE2 for prime order groups is found in [REF]. Note that the choice of M and N is critical for the security proof. The generation method specified in this document is designed to eliminate concerns related to knowing discrete logs of M and N.

SPAKE2+ appears in [TDH] along with a path to a proof that server compromise does not lead to password compromise under the DH assumption (though the corresponding model excludes precomputation attacks).

Elements received from a peer MUST be checked for group membership: failure to properly validate group elements can lead to attacks. Beyond the cofactor multiplication checks to ensure that these elements are in the prime order subgroup of G, 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, and so SPAKE2+ also appears in this document.

7. IANA Considerations

No IANA action is required.

8. 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. Trevor Perrin informed me of SPAKE2+.

9. References

9.1. Normative References

[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>.

[SEC1] SEC, "STANDARDS FOR EFFICIENT CRYPTOGRAPHY, "SEC 1: Elliptic Curve Cryptography", version 2.0", May 2009.

9.2. 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-03 (work in progress), January 2019.

[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>.

[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 5.

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
```

Authors' Addresses

Watson Ladd
UC Berkeley

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu