Workgroup: Network Working Group
Internet-Draft: draft-haase-aucpace-04
Published: 25 July 2021
Intended Status: Informational
Expires: 26 January 2022
Authors: B. Haase
Endress + Hauser Liquid Analysis

(strong) AuCPace, an augmented PAKE

Abstract

This document describes AuCPace which is an augmented PAKE protocol for two parties. The protocol was tailored for constrained devices and smooth migration for compatibility with legacy user credential databases. It is designed to be compatible with any group of both prime- and non-prime order and comes with a security proof providing composability guarantees.

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 26 January 2022.

Table of Contents

1. Introduction

This document describes AuCPace which is an augmented password-authenticated key-establishment (PAKE) protocol for two parties. Both sides the client B and the server A establish a high-entropy session key SK, based on a secret (password) which may be of low entropy for the client B and a password-verifier stored on the server. The protocol is designed such that disclosing the secret to offline dictionary attacks is prevented. Upon server compromise (stealing A's database), the adversary must first succeed with a dictionary search for the clear-text password before being able to impersonate the client.

1.1. Outcome of the CFRG PAKE selection process

AuCPace was one of the two finalists of the CFRG PAKE selection process for the augmented pake protocol use-case in which ultimately OPAQUE was selected as general recommendation of the CFRG working group. OPAQUE and strong AuCPace share the security model and security guarantees but come with specific advantages and drawbacks.

The key advantage of OPAQUE in comparison with AuCPace is that one communication round less than AuCPace is required, allowing for easier integration into TLS 1.3. Applications where the number of communication round-trips are considered critical are encouraged to consider OPAQUE.

The key advantages of AuCPace in comparison are much smaller password verifiers and the possibility to run AuCPace in conjunction with legacy-style password dictionaries that store the conventional triples of (username, salt, password hash). Moreover AuCPace provides a certain level of resilience with respect to adversaries with access to large-scale quantum computers ("quantum annoying property") which OPAQUE does not provide.

1.2. Key design objectives for AuCPace

The AuCPace protocol was specifically tailored for constrained server devices. As such, the computationally complex password hash operation is refered to the clients. AuCPace is also designed for enabling a smooth migration of legacy user credential databases.

AuCPace is designed to be compatible with any group of both prime- and non-prime order and comes with a security proof providing composability guarantees. AuCPace uses CPace as a building block which is described in a separate internet draft document.

AuCPace moreover designed to provide flexibility and a smooth migration process for applications that today don't use a PAKE protocol for authentication but work with conventional password verifier databases instead.

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. Definitions for AuCPace

3.1. Setup

Let C be a group in which there exists a subgroup of prime order p where the computational simultaneous Diffie-Hellman (SDH) problem is hard. C has order p*c where p is a large prime; c will be called the cofactor. Let I be the unit element in C, e.g., the point at infinity in if C is an elliptic curve group. We denote the operations in the group using addition and multiplication operators, e.g. P + (P + P) = P + 2 * P = 3 * P. We refer to a sequence of n additions of an element in P as scalar multiplication by n. With B we denote a generator of the prime-order subgroup in C that we call the base point.

With F we denote a field that may be associated with C, e.g. the prime base field used for representing the coordinates of points on an elliptic curve.

We assume that for any element P in C there is a representation modulo negation, encode_group_element_mod_neg(P) as a byte string such that for any Q in C with Q != P and Q != -P, encode_group_element_mod_neg(P) != encode_group_element_mod_neg(Q). It is recommended that encodings of the elements P and -P share the same result string. Common choices would be a fixed (per-group) length encoding of the x-coordinate of points on an elliptic curve C or its twist C' in Weierstrass form, e.g. according to [IEEE1363] in case of short Weierstrass form curves. For curves in Montgomery form correspondingly the u-coordinate would be encoded, as specified, e.g., by the encodeUCoordinate function from [RFC7748].

With J we denote the group modulo negation associated to C. Note that in J the scalar multiplication operation is well defined since scalar_multiply(P,s) == -scalar_multiply(-P,s) while arbitrary additions of group elements are no longer available.

With J' be denote a second group modulo negation that might share the byte-string encoding function encode_group_element_mod_neg with J such for a given byte string either an element in J or J' is encoded. If the x-coordinate of an elliptic curve point group is used for the encoding, J' would commonly be corresponding to the group of points on the elliptic curve's quadratic twist. Correspondingly, with p' we denote the largest prime factor of the order of J' and its cofactor with c'.

Let scalar_cofactor_clearing(s) be a cofactor clearing function taking an integer input argument and returning an integer as result. For any s, scalar_cofactor_clearing(s) is REQUIRED to be of the form c * s1. I.e. it MUST return a multiple of the cofactor. An example

of such a function may be the cofactor clearing and clamping functions decodeScalar25519 and decodeScalar448 as used in the X25519 and X448 protocols definitions of [RFC7748]. In case of prime-order groups with c == 1, it is RECOMMENDED to use the identity function with scalar_cofactor_clearing(s) = s.

Let scalar_mult_cc(P,s) be a joint "scalar multiplication and cofactor clearing" function of an integer s and an string-encoded value P, where P could represent an element either on J or J'. If P is an element in J or J', the scalar_mult_cc function returns a string encoding of an element in J or J' respectively, such that the result of scalar_mult_cc(P,s) encodes (scalar_cofactor_clearing(s) * P).

Let si = invert_scalar_mult_cc(P,s) be a function such that for any point Z in the prime-order subgroup of J, invert_scalar_mult_cc(scalar_mult_cc(Q,s),s) == Q. A typical implemention will involve calculating the inverse in the prime field mod p on the scalar generated by scalar_cofactor_clearing(s).

Let scalar_mult_ccv(P,s) be a "scalar multiplication cofactor clearing and verify" function of an integer s and an encoding of a group element P. Unlike scalar_mult_cc, scalar_mult_ccv additionally carries out a verification that checks that the computational simultaneous Diffie-Hellman problem (SDH) is hard in the subgroup (in J or J') generated by the encoded element SP = scalar_mult_cc(P,s). In case that the verification fails (SP might be of low order or on the wrong curve), scalar_mult_ccv is REQUIRED to return the encoding of the identity element I. Otherwise scalar_mult_ccv(P,S) is REQUIRED to return the result of scalar_mult_cc(P,s). A common choice for scalar_mult_ccv for Montgomery curves with twist security would be the X25519 and X448 Diffie-Hellman functions as specified in [RFC7748]. For curves in short Weierstrass form, scalar_mult_ccv could be implemented by the combination of a point verification of the input point with a scalar multiplication. Here scalar_mult_ccv SHALL return the encoding of the neutral element I if the input point P was not on the curve C.

Let P=map_to_group_mod_neg(r) be a mapping operation that maps a string r to an encoding of an element P in J. Common choices would be the combination of map_to_base and map_to_curve methods as defined in the hash2curve draft [HASH2CURVE]. Note that we don't require and RECOMMEND cofactor clearing here since this complexity is already included in the definition of the scalar multiplication operation calar_mult_cc above. Additionally requiring cofactor clearing also in map_to_group_mod_neg() would result in efficiency loss.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes. Finally, let nil represent an empty string, i.e., len(nil) = 0.

[f,g,h] denotes alternatives where exactly one of the comma-separated options is to be chosen.

Let H(m) be a hash function from arbitrary strings m to bit strings of a fixed length. Common choices for H are SHA256 or SHA512 [RFC6234]. H is assumed to segment messages m into blocks m_i of byte length H_block. E.g. the blocks used in SHA512 have a size of 128 bytes.

Let strip_sign_information(P) be function that takes a string encoding of an element P in J and strips any information regarding the sign of P, such that strip_sign_information(P) = strip_sign_information(-P). For short Weierstrass (Montgomery) curves this function will return a string encoding the x-coordinate. The purpose of defining this function is for allowing for x-coordinate only scalar multiplication algorithms. The sign is to be stripped before generating the intermediate session key ISK.

With ISK we denote the intermediate session key output string provided by CPace that is generated by a hash operation on the Diffie-Hellman result.

MAC(msg,key) is a message authentication code. Common examples are HMAC_SHA512, a block cipher based MAC or a hash-function based tag.

AEAD(msg,key) is used for denoting an authenticated encryption scheme. A common example would be AES128GCM or Salsa20Poly1305.

KDF(Q) is a key-derivation function that takes a string and derives key of length L. Common choices for KDF are HMAC_SHA512.

With DSI we denote domain-separation identifier strings that may be prepended to the inputs of Hash and KDF functions.

Let IHF(salt, username, pw, sigma) be an iterated hash function that takes a salt value, a user name and a password as input. IHF is designed to slow down brute-force attackers as controlled by a workload parameter set sigma. State of the art iterated hash functions are designed for requiring a large amount of memory for its operation and will be referred to as memory-hard hash functions (MHF). Scrypt [RFC7914] or Argon2 are common examples of a MHF primitive.

With PRS we denote a string that is a required input of the CPace subprotocol and generated by the AuCPace protocol.

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). We denote the parties' representation and the parties themselves both by using the identifiers A and B.

With CI we denote a string that is a required input of the CPace subprotocol. CI is generated together with PRS by the AuCPace augmentation layer. CI SHALL be formed by the concatenation of the identifiers A and B and an associated data string AD, CI = A || B || AD;

With uad we denote an optional string that is specifying user-associated data in a password file entry, such as authorization rights or permissions or account expiration dates. Specification of this string is outside of the scope of the AuCPace protocol.

Let sid be a session id byte string chosen for each protocol session before protocol execution; The length len(sid) SHOULD be larger or equal to 16 bytes.

4. Access to server-side password verifiers databases

AuCPace is an asymmetric PAKE protocol. I.e. while one party, the client B, is in possession of a clear-text password pw, the other party, the server A, is not given access to the cleartext password but only to a password verifier. The way how such a password verifier is maintained by a server party is essential for the AuCPace protocol construction.

4.1. User credential database and password verifier types

With respect to the use of user credantial databases, AuCPace is flexible and supports three different database types.

Firstly, the conventional approach as used for logins without a PAKE protocol is supported (as used e.g. for the password over https:/ approach). Here the server stores a tuple of four elements in his password database file, (username, uad, salt, w) with w = IHF(salt, username, pw, sigma)). With uad, we denote user-accociated data such as permissions. The salt value is a nonce used for the IHF function. Upon a login request, the user transmits the username and the clear-text password pw to the server. The server looks up the database, retrieves the salt value, calculates the IHF function with the given inputs and compares the result with the registered password verifier w. We refer to this setting as "legacy password verifier database" (LPVD) setting.

Secondly, AuCPace supports an AuCPace password verifier database setting (APVD) setting. Here the database is already adapted for use

in conjunction with AuCPace. This setting differs from LPVD by the fact that instead of the direct output of the IHF, w = IHF(salt, username, pw, sigma), an AuCPace password verifier W is maintained in the database. W is calculated from the legacy-style value w by W = scalar_mult_cc(B,w). It is possible to calculate W from a LPVD entry on the fly and without knowledge of the clear-text password upon a remote login request. It is also easily possible to migrate the format of a LPVD database to an APVD database by calling the scalar multiplication function for each entry.

Thirdly, AuCPace supports the *strong* AuCPace password verifier database setting (sAPVD) setting, which differs from APVD and LPVD by the fact that the salt entry in the database, as used for the IHF, is replaced by a salt-derivation parameter q, as will be detailed below. With this strong AuCPace verifier type, the AuCPace protocol provides the additional security guarantee of pre-computation attack resistance. Note that migrating LPVD or APVD database records to sAPVD entries is *not* possible, because calculating the strong sAPVD password verifiers requires the clear-text passwords.

4.2. Encoding of passwords and user names

For AuCPace usernames and passwords are encoded as strings according to the definitions of [RFC8265], i.e. case-preserving unicode encoding SHALL be employed.

4.3. AuCPace database interface for retrieving password verifiers

In the course of the authentication protocoll, the AuCPace server implementation will need to interface to an user credential database for retrieving password verifiers.

Database lookup is implemented by use of string-encoded user names. AuCPace needs an interface equivalent to a function pvr = lookup_pw_verifier_record(username). I.e. a lookup returns a password verfier record pvr, based on a username string. It is REQUIRED that for the purpose of pvr record lookup in databases and for the database contents, the "case preservation" according to [RFC8265] is employed, both for the string encoding of the username and the password.

The content in the pvr records will depend on the application scenarios, LPVD, APVD and sAPVD as defined above. Password verifier

records pvr as returned from the database are REQUIRED to be composed of the following components.

A password verifier from either one of the following two options (w or W).

- LPVD case: A binary encoding of the actual password verifier that has been calculated as a function of the username, the password, a salt value by use of an iterated hash function pv = w = IHF(username,password,salt, sigma).

- APVD and sAPVD cases: An encoding of an element in J, W = scalarmult_cc(B,w), which has been calculated by a scalar multiplication using the base point B. The secret scalar, w, has been calculated by an iterated hash function just as in the case above, w = IHF(username,password,salt, sigma).

sigma, a binary encoding specifying the type and the workload parametrization of the iterated hash function IHF that has been used for calculating w or W.

A salt derivation entry which is either

- LPVD and APVD cases: a binary string encoding of the salt value itself or,

- sAPVD case: an encoding of a secret scalar q for a group scalar multiplication such that the salt value used for calculating w or W could be calculated from the tuple salt = strong_AuCPace_salt_derivation(q,username,password).

In case that a database lookup on a server yields a legacy password database record (LPVD case that includes an entry w), the AuCPace implementation SHALL convert this record into a AuCPace database record (APVD case) by replacing w with W=scalarmult_cc(B,w).

The AuCPace application protocol, thus, only needs to consider the two different APVD and sAPVD scenarios. In case of the APVD scenario, the salt value used for the IHF execution is returned by the database lookup, while in the sAPVD case the salt value is replaced by the secret scalar.

4.4. Derivation of the salt value for strong AuCPace

For strong AuCPace the salt value used for the IHF is not explicitly included in the password verifier record pvr. Instead only the parameter q is stored. q serves for deriving the salt value. In order to determine the salt value, strong AuCPace uses a function

salt = strong_AuCPace_salt_derivation(q,username,password). This function implements the following sequence of operations.

First Z, an element in J, is calculated by use of a Z = map_username_password_to_J(username || password). function, e.g. by use of a hash function such as SHA512 and a mapping such as Elligator2.

The salt value is then determined by applying the scalar q and the group element Z to the scalar multiplication function, salt = strip_sign_information(scalarmult_cc(Z,q)).

Deriving the salt value, thus, requires access to all of, username, clear text password and secret key q. If the database is stolen, an adversary is not able to derive the salt value without having access to the clear-text password.

Note that the method above which derives the salt directly from Z and q will typically only be used when creating a new password database entry. The AuCPace protocol uses a secret scalar r for, masking the value of Z. The approach exploits the relation q * Z == ((Z * r) * q) * (1/r). More explicitly the sequence is as follows:

The client, B, having access to the username and the clear-text password calculates Z. It aims at deriving the salt value.

B samples a fresh scalar r and calculates U = scalarmult_cc(Z,r). B sends U to A.

A fetches q from the database and calculates UQ = scalarmult_cc(U,q). A sends UQ back to A.

B retrieves the salt by inverting the blinding with the scalar r using salt = strip_sign_information(invert_cofactor_cleared_scalar(UQ,r)).

4.5. Specification of the workload parameter sigma

AuCPace does not require the use of a specific iterated hash function IHF. Still it is strongly RECOMMENDED to use AuCPace in conjunction with state-of-the art memory-hard hash functions MHF with a secure workload parametrization. The workload parameter sigma shall encode all of the following.

The algorithm family, such as Argon2i, scrypt or PBKDF2-HMAC-SHA256.

The workload parameter specification, e.g. the iteration count for PBKDF2 or the parameters N,r and p of the scrypt algorithm.

4.6. Result of the database parameter lookup

Summing up, the lookup process for user credential data for AuCPace SHALL provide all of the following information in the course of the session establishment protocol.

The iterated hash function specification sigma.

A salt-derivation parameter and its type. The salt derivationn parameter is either a salt scalar q or the salt value itself. Correspondingly the type of the salt-derivation parameter is either "AuCPace" or "strong AuCPace".

An AuCPace password verifier W = scalarmult_cc(B,IHF(salt, username, pw, sigma)).

For handling the case of failed lookups, if a given user name does not have an entry, the database shall define a default parameter set sigma_default and a default salt-derivation parameter for password hashing. Moreover and a secret string database_seed shall be chosen specifically for each distinct server A. In case of failed lookups the following procedure SHALL be used. A random value w shall be sampled. The password verifier W shall be calculated as W = map_to_group(w). The parameters q or salt shall be respectively calculated as [q,salt] = H(name || database_seed).

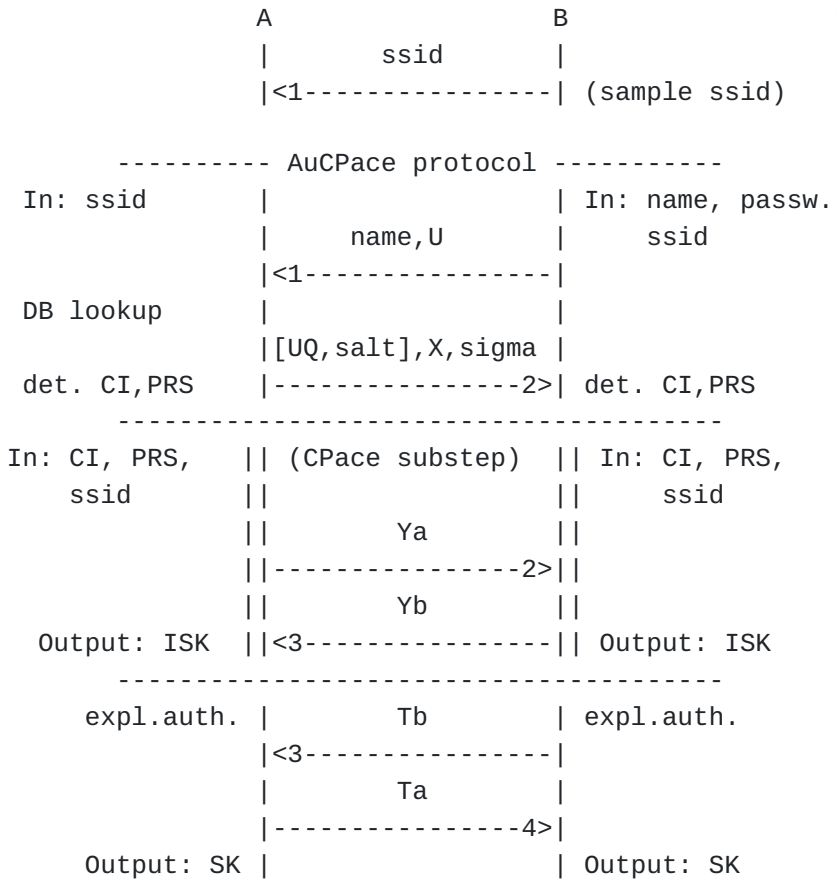
5. Authentication session

5.1. Authentication Protocol Flow

AuCPace is a protocol run between two parties, A and B, for establishing a shared secret key with explicit mutual authentication. AuCPace is implemented by four messages as indicated by the numbers in the figure below. The roles of the two parties are different. Party B, the client, is provided a user name and a password as input. Party A, the server, has access to a user credentials database that stores password verifiers. Both parties share a channel identifier string CI characterizing the communication channel (e.g. information on IP addresses and port numbers of both sides).

The channel identifier, CI, SHOULD include an encoding of the identities of both parties A and B if these are available prior to starting the communication protocol.

Both sides also share a common subsession ID (ssid) string. ssid could be pre-established by a higher-level protocol invocing AuCPace. If no such ssid is available from a higher-level protocol, a suitable approach is including ssid in the first message from B to A as shown in the figure below.



5.2. AuCPace

Both parties start with agreed values on the ssid string. The server side has access to a user credentials database. The client side holds a user name, a clear-text password to be used for the authentication session. Both sides share an encoding CI specifying the communication channel, such as IP addresses and port numbers.

To begin, B calculates an element Z = map_username_password_to_J(username || password) in J.

B then picks r randomly and uniformly according to the requirements for group J scalars and calculates U = scalar_mult_cc(Z,r).

B then sends (name,U) to A.

A then queries its database for the user name and retrieves the information as specified in section Section 4.6 , i.e. the set (W,sigma,salt-derivation parameter [q,salt]). (Note that according to the specification of the database lookup, such a set is provided, even if there is no database entry for the given user name.)

A picks x randomly and uniformly according to the requirements for group J scalaras and calculates X = scalarmult_cc(B,x) and WX =

scalarmult_ccv(W,x). A MUST abort if WX is the neutral element (this indicates an error in the database contents).

A strips the sign information from WX to obtain WXs. A picks ya randomly and uniformly. A then calculates G = map_to_group_mod_neg(DSI1 || WXs || ZPAD || ssid || CI) and Ya = scalar_mult_cc(G,ya). I.e. WXs takes over the role of CPace's PRS string.

The following operations will variate, depending on the type of salt-derivation entry in the database.

   If the salt derivation parameter is for strong AuCPace, A calculates UQ = scalarmult_cc(U,q) and sends (UQ,X,sigma,Ya) to B. B then calculates salt as salt = strip_sign_information(inverse_scalarmult_cc(UQ,r)).

   If the salt derivation parameter is for AuCPace, A sends (salt,X,sigma,Ya) to B.

B then calculates w = IHF(salt, username, pw, sigma) and XW = scalarmult_ccv(X,w). B MUST abort, if XW is the neutral element I.

B strips the sign information from XW to obtain XWs. B picks yb randomly and uniformly. B then calculates G = map_to_group_mod_neg(DSI1 || XWs || ZPAD || ssid || CI) and Yb = scalar_mult_cc(G,yb). B then calculates K = scalar_mult_ccv(Ya,yb). B MUST abort if K is the encoding of the neutral element I. Otherwise B sends Yb to A and proceeds as follows. B strips the sign information from K, Ya and Yb to obtain the strings Ks, Yas and Ybs by using the strip_sign_information() function. B calculates ISK = H(DSI2 || ssid || Ks || Yas || Ybs).

B calculates Tb = MAC(ISK,DSI4) and Ta_v = MAC(ISK,DSI3) and sends (Yb,Tb) to A.

Upon reception of Yb, A calculates K = scalar_mult_ccv(Yb,ya). A MUST abort if K is the neutral element I. If K is different from I, A strips the sign information from K, Ya and Yb and calculates ISK = H(DSI2 || ssid || Ks || Yas || Ybs).

A calculates Tb_v = MAC(ISK,DSI4) and Ta = MAC(ISK,DSI3). If the received authentication tag Tb did not match the verification value Tb_v A MUST abort. Otherwise A sends Ta to B and returns the session key SK = KDF(DSI5 || ISK || ssid).

When B receives T_a it compares this to the verification value Ta_v. B MUST abort if Ta != Ta_v. Otherwise B returns the session key SK = KDF(DSI5 || ISK || ssid).

Upon completion of this protocol, the session key SK returned by A and B will be identical by both parties if and only if the supplied input parameters ssid and CI match on both sides and the password verifier in the server database for the user was calculated from the clear-text password used by B.

6. Authentication of transactions

The AuCPace protocol could also be used for interactive "transactions" that require explicit authentication by use of a password. One example of such a transaction is the request for change of the user's password verifier which involves two components. Firstly, verification of the transaction with the old password and, secondly, the transfer of a payload MT containing a new password verifier for the database.

Another example would be transactions that require special privileges and need explicit password-based authentication (e.g. as in "sudo" commands on unix operating systems).

The common feature of such transactions is the confidential transfer of a transaction payload MT from the client to the server and the authentication of the transaction based on proven knowledge of a password. Upon sucessful authentication, the transaction as specified by MT is processed and a response MR is generated. The server then transfers MR to the client by use of an AEAD scheme.

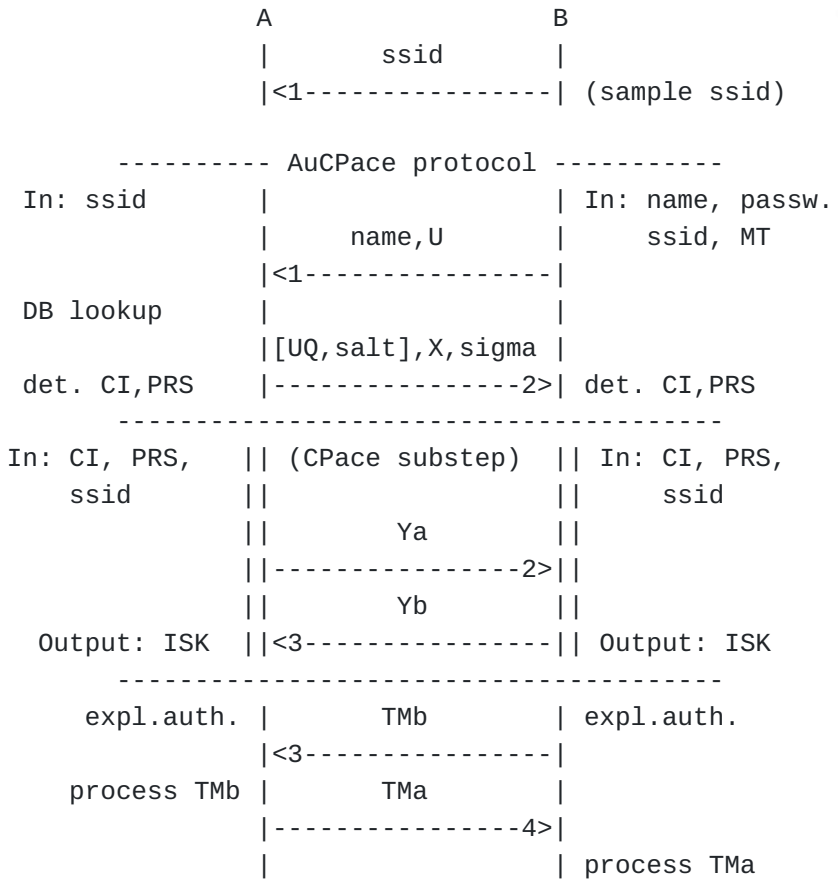
6.1. Transaction Protocol Flow

The protocol flow corresponds to the session key generation above with the difference that the authenticator messages Ta and Tb are replaced by two transaction messages TMa and TMb.

Before starting the protocol, the client has setup a transaction message MT.

As above, the channel identifier, CI, SHOULD include an encoding of the identities of both parties A and B if these are available prior to starting the communication protocol.

Both sides also share a common subsession ID (ssid) string. ssid could be pre-established by a higher-level protocol invocing AuCPace. If no such ssid is available from a higher-level protocol, a suitable approach is including ssid in the first message from B to A as shown in the figure below.



6.2. AuCPace authenticated transactions

The protocol flow is identical to the case of session key generation except for the fact that the authenticators Ta and Tb are replaced by TMa and TMb. TMa is generated by TMb = AEAD(MT,ISK), i.e. the payload MT is encrypted and authenticated by use of ISK. If verification of the authentication tag of TMb succeeds on the server the transaction message is decrypted and executed. E.g. upon a password change transaction message, the new password verifier would be written to the database of the server. The result of the transaction execution is encoded in a response message MR and TMa is calculated as TMa = AEAD(MR,ISK). The client authenticates TMa and returns the decrypted response message MR upon success.

7. Ciphersuites

This section documents AuCPace ciphersuite configurations. A ciphersuite is REQUIRED to specify all of,

- a group modulo negation J with an associated encode_group_element_mod_neg function

- scalar_mult_cc(P,s) and scalar_mult_ccv(P,s) functions operating on encodings of elements P in J

*a mapping function map_to_group_mod_neg(r) converting byte strings r into elements in J

*a strip_sign_information(Q) function operating on string representations of elements Q

*a hash function H for generating the intermediate session key ISK

*A message authentication code MAC for authentication tag generation.

*A key derivation function KDF for generating the final session key SK

*and domain separation strings DSI1, DSI2, DSI3, DSI4, DSI5

Currently, detailed specifications are available for CPACE-X25519-ELLIGATOR2_SHA512-SHA512.

J	map_to_group_mod_neg	KDF
X25519	ELLIGATOR2_SHA512	SHA512 [RFC6234]

Table 1: CPace Ciphersuites

7.1. CPACE-X25519-ELLIGATOR2_SHA512-SHA512

This cipher suite targets particularly constrained targets and implements specific optimizations. It uses the group of points on the Montgomery curve Curve25519 for constructing J. The base field F is the prime field built upon the prime p = 2^255 - 19. The Diffie-Hellmann protocol X25519 and the group are specified in [RFC7748]. The encode_group_element_mod_neg(P) is implemented by the encodeUCoordinate(P) function defined in [RFC7748]. The neutral element I is encoded as a 32 byte zero-filled string.

The domain separation strings are defined as DSI1 = "CPace25519-1", DSI2 = "CPace25519-2", DSI3 = "AuCPace25-Ta", DSI4 = "AuCPace25-Tb", DSI5 = "AuCPace25519". (ASCII encoding without ANSI-C style trailing zeros).

Both, scalar_mult_cc and scalar_mult_ccv, are implemented by the X25519 function specified in [RFC7748].

The secret scalars ya and yb used for X25519 shall be sampled as uniformly distributed 32 byte strings.

The QI = inverse_scalarmult_cc(Q,s) function is implemented as follows.

cs = scalar_cofactor_clearing(s)

csi = 1 / (8 * cs) % p.

QI = UNCLAMPED_X25519(Q, 8 * si).

Where the unclamped X25519 function ommits the setting of bit #254 in the scalar from [RFC7748].

The map_to_group_mod_neg function for the CPace substep is implemented as follows. First the byte length of the ZPAD zero-padding string is determined such that len(ZPAD) = max(0, H_block_SHA512 - len(DSI1 || PRS)), with H_block_SHA512 = 128 bytes. Then a byte string u is calculated by use of u = SHA512(DSI1||PRS||ZPAD||sid||CI). The resulting string is interpreted as 512-bit integer in little-endian format according to the definition of decodeLittleEndian() from [RFC7748]. The resulting integer is then reduced to the base field as input to the Elligator2 map specified in [HASH2CURVE] to yield the secret generator G = Elligator2(u).

The map_to_group_mod_neg function used for the strong AuCPace substep for calculating the field element Z is implemented accordingly. First the byte length of the ZPAD zero-padding string is determined such that len(ZPAD) = max(0, H_block_SHA512 - len(DSI5 || password)), with H_block_SHA512 = 128 bytes. Then a byte string u is calculated by use of u = SHA512(DSI5||password||ZPAD||username). The resulting string is interpreted as 512-bit integer in little-endian format according to the definition of decodeLittleEndian() from [RFC7748]. The resulting integer is then reduced to the base field as input to the Elligator2 map specified in [HASH2CURVE] to yield the secret generator Z = Elligator2(u).

AuCPace25519 calculates an intermediate session key ISK of 64 bytes length by a single invocation of SHA512(DSI2||ISK). Since the encoding does not incorporate the sign from the very beginning Qs = strip_sign_information(Q) == Q for this cipher suite.

AuCPace25519 calculates authentication tags and session key SK from ISK of 64 bytes length by a single invocation of Ta = SHA512(DSI3||ISK), Tb = SHA512(DSI4||ISK) and SK = SHA512(DSI5||ISK).

The following sage code could be used as reference implementation for the mapping and key derivation functions.

```
<CODE BEGINS>

def littleEndianStringToInteger(k):
    bytes = [ord(b) for b in k]
    return sum((bytes[i] << (8 * i)) for i in range(len(bytes)))

def map_to_group_mod_neg_CPace25519(sid, PRS, CI):
    m = hashlib.sha512()
    p = 2^255 - 19

    H_block_SHA512 = 128
    DSI1 = b"CPace25519-1"
    ZPAD_len = max(0,H_block_SHA512 - len(CI) - len(PRS))
    ZPAD = ZPAD_len * "\0"

    m.update(DSI1)
    m.update(PRS)
    m.update(ZPAD)
    m.update(sid)
    m.update(CI)
    u = littleEndianStringToInteger(m.digest())
    return map_to_curve_elligator2_curve25519(u % p)

def map_to_group_mod_neg_StrongAuCPace25519(username, password):
    # Map username and password to field element Z
    DSI = b"AuCPace25519"
    F = GF(2^255 - 19)
    m = hashlib.sha512()
    H_block_SHA512 = 128
    ZPAD_len = max(0,H_block_SHA512 - len(DSI) - len(password))
    ZPAD = ZPAD_len * "\0"
    m.update(DSI)
    m.update(password)
    m.update(ZPAD)
    m.update(username)
    u = littleEndianStringToInteger(m.digest())
    u = F(u)
    Z = map_to_curve_elligator2_curve25519(u)
    return Z

def Inverse_X25519(u_string,scalar_string):
    OrderSubgroup = 2^252 + 27742317777372353535851937790883648493
    SF = GF(OrderSubgroup)
    coFactor = SF(8)
    scalar = clampScalar25519(scalar_string)
    inverse_scalar = 1 /  (SF(scalar) * coFactor)
    inverse_scalar_shifted = Integer(inverse_scalar) * 8
    return X25519(u_string,inverse_scalar_shifted,withClamping=0)
```

```
def generate_ISK_CPace25519(sid,K,Ya,Yb):
    m = hashlib.sha512(b"CPace25519-2")
    m.update(sid)
    m.update(K)
    m.update(Ya)
    m.update(Yb)
    return m.digest()

def MAC_SK_AuCPace25519(ISK):
    DSI3 = b"AuCPace25-Ta"
    DSI4 = b"AuCPace25-Tb"
    DSI5 = b"AuCPace25519"
    m = hashlib.sha512(DSI3)
    m.update(ISK)
    Ta = m.digest()
    Ta = Ta[:16]

    m = hashlib.sha512(DSI4)
    m.update(ISK)
    Tb = m.digest()
    Tb = Tb[:16]

    m = hashlib.sha512(DSI5)
    m.update(ISK)
    SK = m.digest()

    return (Ta,Tb,SK)


<CODE ENDS>
```

Due to its use in Ed25519 [RFC8032], SHA512 is considered to be the natural hash choice for Curve25519. The 512 bit output of SHA512 moreover allows for removing any statistical bias stemming from the non-canonical base field representations, such that the overhead of the HKDF_extract/HKDF_expand sequences from [HASH2CURVE] are considered not necessary (in line with the assessments regarding Curve25519 in [HASH2CURVE]).

8. Security Considerations

A security proof covering AuCPace is found in [HL18].

Elements received from a peer MUST be checked by a proper implementation of the scalar_mult_ccv method. Failure to properly validate group elements can lead to attacks. The Curve25519-based cipher suite employs the twist security feature of the curve for point validation. As such, it is mandatory to check that all low-

order points on both the curve and the twist are mapped on the neutral element by the X25519 function. Corresponding test vectors are provided in the appendix.

The choices of random numbers MUST be uniform. Randomly generated values (e.g., ya, r, q and yb) MUST NOT be reused.

User credential database lookups for AuCPace might not be executed in non-constant time. In this case, AuCPace does not provide full confidentiality with respect to hiding the presence or abscence of a given user's entry in the database. The fact that in case of abscence of a record random values are provided instead should be considered only a mitigation regarding user-enumeration attacks.

If AuCPace is used as a building block of higher-level protocols, it is RECOMMENDED that ssid is generated by the higher-level protocol and passed to AuCPace. It is RECOMMENDED that ssid, is generated by sampling ephemeral random strings.

AuCPace generates the session key SK by a hash function operation on the intermediate session key ISK. If SK is possibly to be used for many different sub-protocols and purposes, such as e.g. in TLS1.3, it is RECOMMENDED to apply SK to a stronger KDF function, such as HKDF from [RFC5869].

In case that side-channel attacks are to be considered practical for a given application, it is RECOMMENDED to focus side-channel protections such as masking and redundant execution (faults) on the process of calculating the secret generator G. The most critical aspect to consider is the processing of the first block of the hash that includes the PRS string. The CPace protocol construction considers the fact that side-channel protections of hash functions might be particularly resource hungry. For this reason, AuCPace aims at minimizing the number of hash functions invocations in the specified mapping method.

AuCPace is designed also for compatibility with legacy-style user-credential databases which directly store the output of iterated hash functions w = IHF(salt, username, pw, sigma). If AuCPace is used in this configuration, AuCPace provides only the security guarantees of a balanced PAKE. I.e. in this case, user impersonation attacks become feasible if the user credential database is stolen. It is RECOMMENDED to migrate legacy-style databases to an AuCPace format, by replacing w with W = scalarmult_cc(B,w).

It is RECOMMENDED to use password verifiers for "strong AuCPace" on such a migrated database upon user password changes in order to obtain pre-computation attack resistance.

The Map2Point primitive used for (strong) AuCPace and for the CPace substep needs to be be probabilistically invertible according to the definition of [CPaceAnalysis]. Use of Elligator2 (for Montgomery or Edwards curves) or simplified SWU [HASH2CURVE] (otherwise) is recommended for this purpose.

AuCPace is proven secure under the hardness of the strong computational simultaneous Diffie-Hellmann (SDH) problem in the group J as defined in [CPaceAnalysis]. Still, even for the event that large-scale quantum computers (LSQC) will become available, AuCPace forces an active adversary to solve one instance of the CDH problem per password guess. Using the wording suggested by Steve Thomas on the CFRG mailing list, AuCPace is "quantum-annoying".

Strong AuCPace is pre-computation attack resistant under the CDH and the gap One-More-DH assumption, modelling the hash2curve primitive as random oracle hashing to the group. If the map2point primitive used is probabilistically invertible, the analysis from [CPaceAnalysis] applies for instantiating the OPRF used in strong AuCPace also using single-coordinate Diffie-Hellman using X25519 and inverse X25519.

While the rest of the AuCPace protocol is quantum-annoying, solving a single discrete logarithm problem will reveal the salt-derivation parameter q as used in strong AuCPace to the adversary. This will allow for pre-computing a rainbow table for a given user name. I.e. the pre-computation attack resistance property will be lost and security guarantees of strong AuCPace and AuCPace would become equivalent. (In order maintain strong AuCPace's pre-computation attack guarantees, a post-quantum instance of an olibvious pseudo-random function (OPRF) would have to be found and used as replacement of the Diffie-Hellman OPRF construction employed by strong AuCPace.)

9. IANA Considerations

No IANA action is required.

10. Acknowledgments

Thanks to the members of the CFRG for comments and advice.

11. References

11.1. Normative References

[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>.

[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>.

[HASH2CURVE] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "draft-irtf-cfrg-hash-to-curve-05", 2019. IRTF draft standard

[IEEE1363] IEEE, ""Standard Specifications for Public Key Cryptography", IEEE 1363", 2000. IEEE 1363

11.2. Informative References

[HL18] Haase, B. and B. Labrique, "AuCPace. PAKE protocol tailored for the use in the internet of things.", February 2018. eprint.iacr.org/2018/286

[CPaceAnalysis] Abdalla, M., Haase, B., and J. Hesse, "Security analysis of CPace.", January 2021. eprint.iacr.org/2021/114

[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>.

Appendix A. AuCPace25519 Test Vectors

A.1. Inverse X25519 test vectors

```
######################## /inverse X25519 ##########################

Z :
0x5d7189be6192ffccdb80902ac26c5d38592822a761c7268007200a232a4cd841
r (random scalar input for X25519):
0xe8a70b556490ecdbd52c2927464d4eff557807496af234fc496c9f4221bd4423
U = X25519(Z,r):
0x3e6377b3410a60d0ca65190963ad6dce3aeae178aafad369dc2a59c59acc3ceb
IU = inverse_X25519(U)
0x5d7189be6192ffccdb80902ac26c5d38592822a761c7268007200a232a4cd841

Z :
0x73583bc520f4b9e1245e29b92e6b12dc1c8bc0d9b018a1e2626875db2774974
r (random scalar input for X25519):
0xe22cb510bd45fc6cbcb929353777925d152c2a9a5b924715d7480aad8b64d447
U = X25519(Z,r):
0x545040a9ac1cc13a627dddc0ab4ef0b9128d54476aa98727bd45a86ea2d6de24
IU = inverse_X25519(U)
0x73583bc520f4b9e1245e29b92e6b12dc1c8bc0d9b018a1e2626875db2774974

######################## inverse X25519/ ##########################
```

A.2. Strong AuCPace25519 salt test vectors

```
############ /salt derivation for strong AuCPace ##################

DSI5 = 417543506163653235353139 string ('AuCPace25519') of len(12)
pw   = 70617373776f7264 ('password') string of len(8)
ZPAD = 108 zero bytes
name = 757365726e616d65 ('username') string of len(8)
u = SHA512(DSI||password||ZPAD||username) as 512 bit int:
 0xfec497749d249426e4895e05d0bb4f565aa4423f33d19e6b20aa3837eb77d16e
 <<256
 + 0xb8f5b1b51c1557c088356d7ca09bd78f259f1d5f4041d466f4edd40f041a0bb3

u as reduced base field element coordinate:
  0xa242d046f835586749962599c699e609a00f2c0f15f584dce322c5bf7e327be

Z = Elligator2(u) as base field element coordinate:
  0x3c01681e4c6d4a43b527c789bcf6046b53ac14c516dfbbbb0f8916826b537f4b

salt-derivation parameter q:
0xf40546b4544bae5fcc564dc917407ee02300312c8fd558a0b37f48322277962e

ZQ = X25519(Z,q):
0x77412819cad958e14d4a4c09c129456b90733fd13f337ffed6c0a30f7c3a9a50


Blinding scalar r:
0x1698d57693a684a957ae0492bade0dfd6a261dfa2bb4a74c6b0b8b84acf082a8
U = X25519(Z,r):
0xaf2a058fbd974a6122f8316323060d8087057019716661214 77eba97386a977
UQ = X25519(U,q):
0x610270daff540b5b7adec057e0f8fa1bfe7687649d95f65560a77a2be70e6cb5
ZQ = inverse_X25519(Z,r)
0x77412819cad958e14d4a4c09c129456b90733fd13f337ffed6c0a30f7c3a9a50

############ salt derivation for strong AuCPace/ ##################
```

A.3. Test vectors for AuCPace password verifier

```
###################### /Password verifier #########################
Inputs:

Password: 'password', length 8
User Name: 'username', length 8

Z for username, password:
0x3c01681e4c6d4a43b527c789bcf6046b53ac14c516dfbbbb0f8916826b537f4b
q:
0xf40546b4544bae5fcc564dc917407ee02300312c8fd558a0b37f48322277962e

Salt value salt = X25519(Z,q):
0x77412819cad958e14d4a4c09c129456b90733fd13f337ffed6c0a30f7c3a9a50

######################################################################

Concatenated input to scrypt as password parameter:
'passwordusername', length 16

Salt value Z^q:
0x77412819cad958e14d4a4c09c129456b90733fd13f337ffed6c0a30f7c3a9a50
Password hash w from scrypt with N=1<<15, r=8, p=1
0xd832084ac895c20b4893ba4541daeb41ee8ae6cf99788ac8fda4a125734eb5f2
Password verifier W = X25519(B,w)
0x12511fcfc70fe9c3a8e72b347d7de52927fc253b83edc8271a5e90ecdf958f57
Server private key x =
0x8b2389da423eb4f8979245d5b4527f92eacc1f3ad11581822d56498c44d4aba4
Server Public key X = X25519(B,x)
0x3e97ae1689d3b1c9fbd82e0cc036118f44279f58537a6b13b2db87e62682afd7
Shared secret XW = X25519(W,x)
0x3e97ae1689d3b1c9fbd82e0cc036118f44279f58537a6b13b2db87e62682afd7
###################### Password verifier/ #########################
```

Author's Address

Bjoern Haase
Endress + Hauser Liquid Analysis

Email: bjoern.m.haase@web.de