PSA Cryptography API 1.1 PAKE Extension

Document number:	AES 0058
Release Quality:	Beta
Issue Number:	0
Confidentiality:	Non-confidential
Date of Issue:	10/02/2022

BETA RELEASE

This is a proposed update to the *IHI 0086 PSA Cryptography API 1.1* specification.

This is a BETA release in order to enable wider review and feedback on the changes proposed to be included in a future version of the specification.

At this quality level, the proposed changes and interfaces are complete, and suitable for initial product development. However, the specification is still subject to change.

Abstract

This document is part of the PSA family of specifications. It defines an extension to the PSA Cryptography API, to introduce support for Password-authenticated key exchange (PAKE) algorithms.

Contents

About this document

Release information

The change history table lists the changes that have been made to this document.

Date	Version	Confidentiality	Change
February 2022	Beta 0	Non-confidential	Initial release of the 1.1 PAKE Extension specification

PSA Cryptography API

Arm Non-Confidential Document Licence (“Licence”)

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

References

This document refers to the following documents.

Ref	Document Number	Title
[PSA-CRYPT]	IHI 0086	*PSA Cryptography API*. https://developer.arm.com/documentation/ihi0086/latest
[MBED-TLS]		Arm Ltd, *Mbed TLS*. https://github.com/ARMmbed/mbedtls
[SEC1]		Standards for Efficient Cryptography, *SEC 1: Elliptic Curve Cryptography*, May 2009. https://www.secg.org/sec1-v2.pdf
[RFC8235]		IETF, *Schnorr Non-interactive Zero-Knowledge Proof*, September 2017. https://tools.ietf.org/html/rfc8235.html
[RFC8236]		IETF, *J-PAKE: Password-Authenticated Key Exchange by Juggling*, September 2017. https://tools.ietf.org/html/rfc8236.html

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AEAD	See *Authenticated Encryption with Associated Data*.
Algorithm	A finite sequence of steps to perform a particular operation. In this specification, an algorithm is a *cipher* or a related function. Other texts call this a cryptographic mechanism.
API	Application Programming Interface.
Asymmetric	See *Public-key cryptography*.
Authenticated Encryption with Associated Data (AEAD)	A type of encryption that provides confidentiality and authenticity of data using *symmetric* keys.
Byte	In this specification, a unit of storage comprising eight bits, also called an octet.
Cipher	An algorithm used for encryption or decryption with a *symmetric* key.
Cryptoprocessor	The component that performs cryptographic operations. A cryptoprocessor might contain a *keystore* and countermeasures against a range of physical and timing attacks.
Hash	A cryptographic hash function, or the value returned by such a function.

Table 2 (continued)

Term	Meaning
HMAC	A type of *MAC* that uses a cryptographic key with a *hash* function.
IMPLEMENTATION DEFINED	Behavior that is not defined by the architecture, but is defined and documented by individual implementations.
Initialization vector (IV)	An additional input that is not part of the message. It is used to prevent an attacker from making any correlation between cipher text and plain text. This specification uses the term for such initial inputs in all contexts. For example, the initial counter in CTR mode is called the IV.
IV	See *Initialization vector*.
KDF	See *Key Derivation Function*.
Key agreement	An algorithm for two or more parties to establish a common secret key.
Key Derivation Function (KDF)	Key Derivation Function. An algorithm for deriving keys from secret material.
Key identifier	A reference to a cryptographic key. Key identifiers in the PSA Crypto API are 32-bit integers.
Key policy	Key metadata that describes and restricts what a key can be used for.
Key size	The size of a key as defined by common conventions for each key type. For keys that are built from several numbers of strings, this is the size of a particular one of these numbers or strings. This specification expresses key sizes in bits.
Key type	Key metadata that describes the structure and content of a key.
Keystore	A hardware or software component that protects, stores, and manages cryptographic keys.
Lifetime	Key metadata that describes when a key is destroyed.
MAC	See *Message Authentication Code*.
Message Authentication Code (MAC)	A short piece of information used to authenticate a message. It is created and verified using a *symmetric* key.
Message digest	A *hash* of a message. Used to determine if a message has been tampered.
Multi-part operation	An *API* which splits a single cryptographic operation into a sequence of separate steps.
Non-extractable key	A key with a *key policy* that prevents it from being read by ordinary means.
Nonce	Used as an input for certain *AEAD* algorithms. Nonces must not be reused with the same key because this can break a cryptographic protocol.
PAKE	See *Password-authenticated key exchange*.

Table 2 (continued)

Term	Meaning
Password-authenticated key exchange (PAKE)	An interactive method for two or more parties to establish cryptographic keys based on knowledge of a low entropy secret, such as a password. This can provide strong security for communication from a weak password, because the password is not directly communicated as part of the key exchange.
Persistent key	A key that is stored in protected non-volatile memory.
PSA	Platform Security Architecture
Public-key cryptography	A type of cryptographic system that uses key pairs. A keypair consists of a (secret) private key and a public key (not secret). A public key cryptographic algorithm can be used for key distribution and for digital signatures.
Salt	Used as an input for certain algorithms, such as key derivations.
Signature	The output of a digital signature scheme that uses an *asymmetric* keypair. Used to establish who produced a message.
Single-part function	An *API* that implements the cryptographic operation in a single function call.
SPECIFICATION DEFINED	Behavior that is defined by this specification.
Symmetric	A type of cryptographic algorithm that uses a single key. A symmetric key can be used with a block cipher or a stream cipher.
Volatile key	A key that has a short lifespan and is guaranteed not to exist after a restart of an application instance.

Conventions

Typographical conventions

The typographical conventions are:

italic	Introduces special terminology, and denotes citations.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
SMALL CAPITALS	Used for some common terms such as IMPLEMENTATION DEFINED. Used for a few terms that have specific technical meanings, and are included in the *Terms and abbreviations*.
Red text	Indicates an open issue.
Blue text	Indicates a link. This can be • A cross-reference to another location within the document • A URL, for example http://infocenter.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Current status and anticipated changes

This document is at Beta quality status which has a particular meaning to Arm of which the recipient must be aware. A Beta quality specification will be sufficiently stable & committed for initial product development, however all aspects of the architecture described herein remain SUBJECT TO CHANGE. Please ensure that you have the latest revision.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to arm.psa-feedback@arm.com. Give:

- The title (PSA Cryptography API).
- The number and issue (AES 0058 1.1 PAKE Extension Beta (Issue 0)).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1 Introduction

This document introduces an extension to the *PSA Cryptography API* [PSA-CRYPT] specification, to provide support for *Password-authenticated key exchange* (PAKE) algorithms, and specifically for the J-PAKE algorithm.

When the proposed extension is sufficiently stable to be classed as Final, it will be integrated into a future version of [PSA-CRYPT].

This specification must be read and implemented in conjunction with [PSA-CRYPT]. All of the conventions, design considerations, and implementation considerations that are described in [PSA-CRYPT] apply to this specification.

> **Note:**
>
> This extension has been developed in conjunction with the *Mbed TLS* [MBED-TLS] project, which is developing an implementation of the PSA Cryptography API.

Note

This version of the document includes *Rationale* commentary that provides background information relating to the design decisions that led to the current proposal. This enables the reader to understand the wider context and alternative approaches that have been considered.

The rationale is presented in green boxes, as this note is.

1.1 Objectives for the PAKE Extension

1.1.1 Scheme review

There are a number of PAKE protocols in circulation, but none of them are used widely in practice, and they are very different in scope and mechanics. The API proposed for the PSA Cryptography API focuses on schemes that are most likely to be needed by users. A number of factors are used to identify important PAKE algorithms.

Wide deployment

Considering PAKE schemes with already wide deployment allows users with existing applications to migrate to PSA. Currently there is only one scheme with non-negligible success in the industry: Secure Remote Password (SRP).

Requests

Some PAKE schemes have been requested by the community and need to be supported. Currently, these are SPAKE2+ and J-PAKE (in particular the Elliptic Curve based variant, sometimes known as ECJPAKE)

Standardization

There are PAKE schemes that are being standardized and will be recommended for use in future protocols. To ensure that the API is future proof, we need to consider these. The CFRG recommends CPace and OPAQUE for use in IETF protocols. These are also recommended for use in TLS and IKE in the future.

Applications

Some of these schemes are used in popular protocols. This information confirms the choices already made and can help to extend the list in future:

PAKE scheme	Protocols
J-PAKE	TLS, THREAD v1
SPAKE2+	CHIP
SRP	TLS
OPAQUE	TLS, IKE
CPace	TLS, IKE
Dragonfly	WPA3 (Before including the Dragonblood attack should be considered as well.)
SPAKE	Kerberos 5 v1.17
PACE	IKEv2
AugPAKE	IKEv2

1.1.2 Scope of the PAKE Extension

The following PAKE schemes are considered in the PSA Crypto API design:

Balanced	Augmented
J-PAKE	SRP
SPAKE2	SPAKE2+
CPace	OPAQUE

Scope of this specification

The current API proposal provides the general interface for PAKE algorithms, and the specific interface for J-PAKE.

Out of scope

PAKE protocols that do not fit into any of the above categories are not taken into consideration in the proposed API. Some schemes like that are:

PAKE scheme	Specification
AMP	IEEE 1363.2, ISO/IEC 11770-4
BSPEKE2	IEEE 1363.2
PAKZ	IEEE 1363.2
PPK	IEEE 1363.2
SPEKE	IEEE 1363.2
WSPEKE	IEEE 1363.2
SPEKE	IEEE 1363.2
PAK	IEEE 1363.2, X.1035, RFC 5683
EAP-PWD	RFC 5931
EAP-EKE	RFC 6124
IKE-PSK	RFC 6617
PACE for IKEv2	RFC 6631
AugPAKE for IKEv2	RFC 6628
PAR	IEEE 1363.2
SESPAKE	RFC 8133
ITU-T	X.1035
SPAKE1	
Dragonfly	
B-SPEKE	
PKEX	
EKE	
Augmented-EKE	
PAK-X	
PAKE	

The exception is SPAKE2, because of it is related to SPAKE2+.

2 Password-authenticated key exchange (PAKE)

This is a proposed PAKE interface for *PSA Cryptography API* [PSA-CRYPT]. It is not part of the official PSA Cryptography API yet.

> **Note:**
>
> The content of this specification is not part of the stable PSA Cryptography API and may change substantially from version to version.

2.1 Algorithm encoding

A new algorithm category is added for PAKE algorithms. The algorithm category table in [PSA-CRYPT] Appendix B is extended with the information in Table 3.

Table 3 New algorithm identifier categories

Algorithm category	CAT	Category details
PAKE	`0x0A`	See *PAKE algorithm encoding*

2.1.1 PAKE algorithm encoding

The algorithm identifier for PAKE algorithms defined in this specification are encoded as shown in Figure 1.

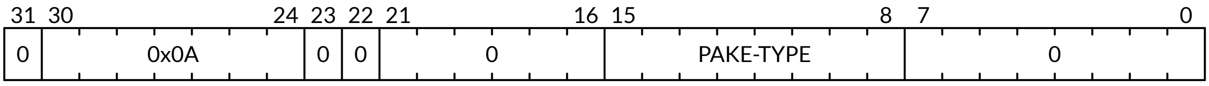

Figure 1 PAKE algorithm encoding

The defined values for PAKE-TYPE are shown in Table 4.

Table 4 PAKE algorithm sub-type values

PAKE algorithm	PAKE-TYPE	Algorithm identifier	Algorithm value
J-PAKE	`0x01`	`PSA_ALG_JPAKE`	`0x0A000100`

2.2 Changes and additions to the Programming API

2.2.1 PAKE algorithms

`PSA_ALG_IS_PAKE` (macro)

Whether the specified algorithm is a password-authenticated key exchange.

```
#define PSA_ALG_IS_PAKE(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier: a value of type psa_algorithm_t.

Returns

1 if alg is a password-authenticated key exchange (PAKE) algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_JPAKE (macro)

The Password-authenticated key exchange by juggling (J-PAKE) algorithm.

```
#define PSA_ALG_JPAKE ((psa_algorithm_t)0x0a000100)
```

This is J-PAKE as defined by *J-PAKE: Password-Authenticated Key Exchange by Juggling* [RFC8236], instantiated with the following parameters:

- The group can be either an elliptic curve or defined over a finite field.
- Schnorr NIZK proof as defined by *Schnorr Non-interactive Zero-Knowledge Proof* [RFC8235], using the same group as the J-PAKE algorithm.
- A cryptographic hash function.

To select these parameters and set up the cipher suite, initialize a psa_pake_cipher_suite_t object, and call the following functions in any order:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(cipher_suite, PSA_ALG_JPAKE);
psa_pake_cs_set_primitive(cipher_suite,
                          PSA_PAKE_PRIMITIVE(type, family, bits));
psa_pake_cs_set_hash(cipher_suite, hash);
```

More information on selecting a specific Elliptic curve or Diffie-Hellman field is provided with the PSA_PAKE_PRIMITIVE_TYPE_ECC and PSA_PAKE_PRIMITIVE_TYPE_DH constants.

The J-PAKE operation follows the protocol shown in Figure 2 on page 14.

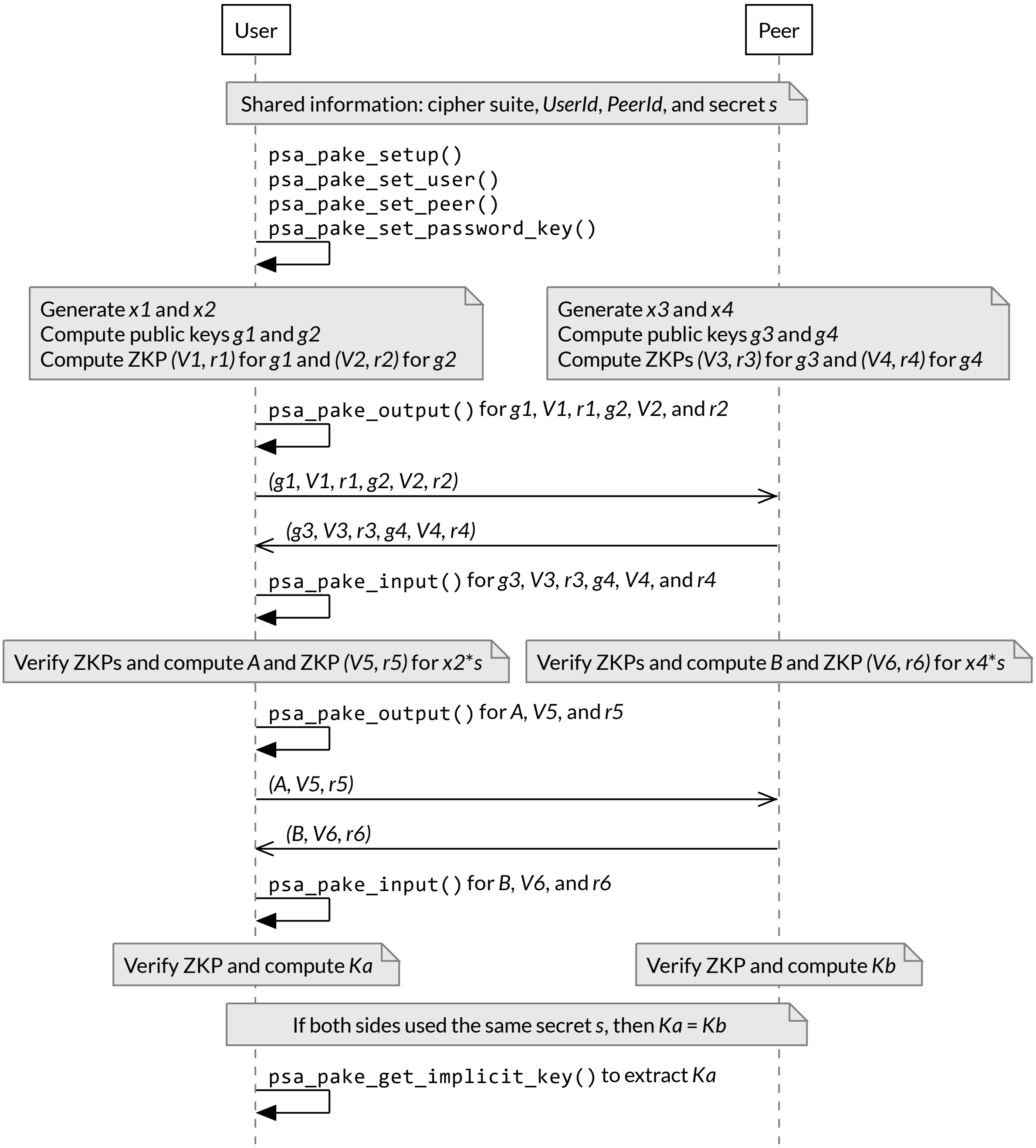
User
Peer
Shared information: cipher suite, *UserId*, *PeerId*, and secret *s*
psa_pake_setup()
psa_pake_set_user()
psa_pake_set_peer()
psa_pake_set_password_key()
Generate *x1* and *x2*
Compute public keys *g1* and *g2*
Compute ZKP *(V1, r1)* for *g1* and *(V2, r2)* for *g2*
Generate *x3* and *x4*
Compute public keys *g3* and *g4*
Compute ZKPs *(V3, r3)* for *g3* and *(V4, r4)* for *g4*
psa_pake_output() for *g1, V1, r1, g2, V2*, and *r2*
(g1, V1, r1, g2, V2, r2)
(g3, V3, r3, g4, V4, r4)
psa_pake_input() for *g3, V3, r3, g4, V4*, and *r4*
Verify ZKPs and compute *A* and ZKP *(V5, r5)* for *x2*s*
Verify ZKPs and compute *B* and ZKP *(V6, r6)* for *x4*s*
psa_pake_output() for *A, V5*, and *r5*
(A, V5, r5)
(B, V6, r6)
psa_pake_input() for *B, V6*, and *r6*
Verify ZKP and compute *Ka*
Verify ZKP and compute *Kb*
If both sides used the same secret *s*, then *Ka* = *Kb*
psa_pake_get_implicit_key() to extract *Ka*

Figure 2 The J-PAKE protocol.

The variable names *x1*, *g1*, and so on, are taken from the finite field implementation of J-PAKE in [RFC8236] §2. Details of the computation for the key shares and zero-knowledge proofs are in [RFC8236] and [RFC8235].

J-PAKE does not assign roles to the participants, so it is not necessary to call `psa_pake_set_role()`.

J-PAKE requires both an application and a peer identity. If the peer identity provided to `psa_pake_set_peer()` does not match the data received from the peer, then the call to `psa_pake_input()` for the `PSA_PAKE_STEP_ZK_PROOF` step will fail with `PSA_ERROR_INVALID_SIGNATURE`.

The following steps demonstrate the application code for ‘User’ in Figure 2. The input and output steps must be carried out in exactly the same sequence as shown.

1. To prepare a J-Pake operation, initialize and set up a psa_pake_operation_t object by calling the following functions:

   ```
   psa_pake_operation_t jpake = PSA_PAKE_OPERATION_INIT;

   psa_pake_setup(&jpake, &cipher_suite);
   psa_pake_set_user(&jpake, ...);
   psa_pake_set_peer(&jpake, ...);
   psa_pake_set_password_key(&jpake, ...);
   ```

   The password is provided as a key. This can be the password text itself, in an agreed character encoding, or some value derived from the password as required by a higher level protocol.

   The key material is used as an array of bytes, which is converted to an integer as described in *SEC 1: Elliptic Curve Cryptography* [SEC1] §2.3.8, before reducing it modulo *q*. Here, *q* is the order of the group defined by the cipher-suite primitive. psa_pake_set_password_key() will return an error if the result of the conversion and reduction is 0.

After setup, the key exchange flow for J-PAKE is as follows:

2. To get the first round data that needs to be sent to the peer, call:

   ```
   // Get g1
   psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
   // Get V1, the ZKP public key for x1
   psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
   // Get r1, the ZKP proof for x1
   psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
   // Get g2
   psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
   // Get V2, the ZKP public key for x2
   psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
   // Get r2, the ZKP proof for x2
   psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
   ```

3. To provide the first round data received from the peer to the operation, call:

   ```
   // Set g3
   psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
   // Set V3, the ZKP public key for x3
   psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
   // Set r3, the ZKP proof for x3
   psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
   // Set g4
   psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
   // Set V4, the ZKP public key for x4
   psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
   // Set r4, the ZKP proof for x4
   psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
   ```

4. To get the second round data that needs to be sent to the peer, call:

   ```
   // Get A
   psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
   ```

(continues on next page)

(continued from previous page)

```
// Get V5, the ZKP public key for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r5, the ZKP proof for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

5. To provide the second round data received from the peer to the operation call:

```
// Set B
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V6, the ZKP public key for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r6, the ZKP proof for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

6. To use the shared secret, set up a key derivation operation and transfer the computed value:

```
// Set up the KDF
psa_key_derivation_operation_t kdf = PSA_KEY_DERIVATION_OPERATION_INIT;
psa_key_derivation_setup(&kdf, ...);
psa_key_derivation_input_bytes(&kdf, PSA_KEY_DERIVATION_INPUT_CONTEXT, ...);
psa_key_derivation_input_bytes(&kdf, PSA_KEY_DERIVATION_INPUT_LABEL, ...);

// Get Ka=Kb=K
psa_pake_get_implicit_key(&jpake, &kdf)
```

For more information about the format of the values which are passed for each step, see *PAKE step types* on page 24.

If the verification of a Zero-knowledge proof provided by the peer fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_ZK_PROOF` step will return `PSA_ERROR_INVALID_SIGNATURE`.

Warning: At the end of this sequence there is a cryptographic guarantee that only a peer that used the same password is able to compute the same key. But there is no guarantee that the peer is the participant it claims to be, or that the peer used the same password during the exchange.

At this point, authentication is implicit — material encrypted or authenticated using the computed key can only be decrypted or verified by someone with the same key. The peer is not authenticated at this point, and no action should be taken by the application which assumes that the peer is authenticated, for example, by accessing restricted files.

To make the authentication explicit, there are various methods to confirm that both parties have the same key. See [RFC8236] §5 for two examples.

Compatible key types

`PSA_KEY_TYPE_PASSWORD`

`PSA_KEY_TYPE_PASSWORD_HASH`

2.2.2 PAKE primitives

A PAKE algorithm specifies a sequence of interactions between the participants. Many PAKE algorithms are designed to allow different cryptographic primitives to be used for the key establishment operation, so long as all the participants are using the same underlying cryptography.

The cryptographic primitive for a PAKE operation is specified using a psa_pake_primitive_t value, which can be constructed using the PSA_PAKE_PRIMITIVE() macro, or can be provided as a numerical constant value.

A PAKE primitive is required when constructing a PAKE cipher-suite object, psa_pake_cipher_suite_t, which fully specifies the PAKE operation to be carried out.

psa_pake_primitive_type_t (type)

Encoding of the type of the PAKE's primitive.

```
typedef uint8_t psa_pake_primitive_type_t;
```

The range of PAKE primitive type values is divided as follows:

0x00 Reserved as an invalid primitive type.

0x01 – 0x7f Specification-defined primitive type. Primitive types defined by this standard always have bit 7 clear. Unallocated primitive type values in this range are reserved for future use.

0x80 – 0xff Implementation-defined primitive type. Implementations that define additional primitive types must use an encoding with bit 7 set.

For specification-defined primitive types, see the documentation of individual PSA_PAKE_PRIMITIVE_TYPE_XXX constants.

PSA_PAKE_PRIMITIVE_TYPE_ECC (macro)

The PAKE primitive type indicating the use of elliptic curves.

```
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
```

The values of the family and bits components of the PAKE primitive identify a specific elliptic curve, using the same mapping that is used for ECC keys. See the definition of psa_ecc_family_t. Here family and bits refer to the values used to construct the PAKE primitive using PSA_PAKE_PRIMITIVE().

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys on the specific Elliptic curve. For more information, consult the documentation of psa_export_public_key().
- The format for scalars is the same as that for private keys on the specific Elliptic curve. For more information, consult the documentation of psa_export_key().

PSA_PAKE_PRIMITIVE_TYPE_DH (macro)

The PAKE primitive type indicating the use of Diffie-Hellman groups.

```
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
```

The values of the family and bits components of the PAKE primitive identify a specific Diffie-Hellman group, using the same mapping that is used for Diffie-Hellman keys. See the definition of psa_dh_family_t. Here family and bits refer to the values used to construct the PAKE primitive using PSA_PAKE_PRIMITIVE().

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys in the specific Diffie-Hellman group. For more information, consult the documentation of psa_export_public_key().
- The format for scalars is the same as that for private keys in the specific Diffie-Hellman group. For more information, consult the documentation of psa_export_key().

psa_pake_family_t (type)

Encoding of the family of the primitive associated with the PAKE.

```
typedef uint8_t psa_pake_family_t;
```

For more information see the documentation of individual PSA_PAKE_PRIMITIVE_TYPE_XXX constants.

psa_pake_primitive_t (type)

Encoding of the primitive associated with the PAKE.

```
typedef uint32_t psa_pake_primitive_t;
```

PAKE primitive values are constructed using PSA_PAKE_PRIMITIVE().

Rationale

An integral type is required for psa_pake_primitive_t to enable values of this type to be compile-time-constants. This allows them to be used in case statements, and used to calculate static buffer sizes with PSA_PAKE_OUTPUT_SIZE() and PSA_PAKE_INPUT_SIZE().

PSA_PAKE_PRIMITIVE (macro)

Construct a PAKE primitive from type, family and bit-size.

```
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
```

Parameters

pake_type — The type of the primitive: a value of type psa_pake_primitive_type_t.

pake_family — The family of the primitive. The type and interpretation of this parameter depends on pake_type. For more information, consult the documentation of individual psa_pake_primitive_type_t constants.

`pake_bits`

The bit-size of the primitive: a value of type `size_t`. The interpretation of this parameter depends on `family`. For more information, consult the documentation of individual `psa_pake_primitive_type_t` constants.

Returns: `psa_pake_primitive_t`

The constructed primitive value. Return `0` if the requested primitive can't be encoded as `psa_pake_primitive_t`.

2.2.3 PAKE cipher suites

A PAKE algorithm uses a specific cryptographic primitive for key establishment, specified using a PAKE primitive. PAKE algorithms also require a cryptographic hash algorithm, which is agreed between the participants.

The `psa_pake_cipher_suite_t` object is used to fully specify a PAKE operation, combining the PAKE algorithm, the PAKE primitive, the hash or any other algorithm that parametrises the PAKE in question.

A PAKE cipher suite is required when setting up a PAKE operation in `psa_pake_setup()`.

`psa_pake_cipher_suite_t` (type)

The type of an object describing a PAKE cipher suite.

```
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
```

This is the object that represents the cipher suite used for a PAKE algorithm. The PAKE cipher suite specifies the PAKE algorithm, and the options selected for that algorithm. The cipher suite includes the following attributes:

- The PAKE algorithm itself.
- The PAKE primitive, which identifies the prime order group used for the key exchange operation. See *PAKE primitives* on page 17.
- The hash algorithm to use in the operation.

> **Note:**
>
> Implementations are recommended to define the cipher-suite object as a simple data structure, with fields corresponding to the individual cipher suite attributes. In such an implementation, each function `psa_pake_cs_set_xxx()` sets a field and the corresponding function `psa_pake_cs_get_xxx()` retrieves the value of the field.
>
> An implementations can report attribute values that are equivalent to the original one, but have a different encoding. For example, an implementation can use a more compact representation for attributes where many bit-patterns are invalid or not supported, and store all values that it does not support as a special marker value. In such an implementation, after setting an invalid value, the corresponding get function returns an invalid value which might not be the one that was originally stored.

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

Before calling any function on a PAKE cipher suite object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite;
  memset(&cipher_suite, 0, sizeof(cipher_suite));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_pake_cipher_suite_t cipher_suite;
  ```

- Initialize the object to the initializer `PSA_PAKE_CIPHER_SUITE_INIT`, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;
  ```

- Assign the result of the function `psa_pake_cipher_suite_init()` to the object, for example:

  ```
  psa_pake_cipher_suite_t cipher_suite;
  cipher_suite = psa_pake_cipher_suite_init();
  ```

PSA_PAKE_CIPHER_SUITE_INIT (macro)

This macro returns a suitable initializer for a PAKE cipher suite object of type `psa_pake_cipher_suite_t`.

```
#define PSA_PAKE_CIPHER_SUITE_INIT /* implementation-defined value */
```

psa_pake_cipher_suite_init (function)

Return an initial value for a PAKE cipher suite object.

```
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
```

Returns: `psa_pake_cipher_suite_t`

psa_pake_cs_get_algorithm (function)

Retrieve the PAKE algorithm from a PAKE cipher suite.

```
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_algorithm_t`

The PAKE algorithm stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_algorithm` (function)

Declare the PAKE algorithm for the cipher suite.

```
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`alg`	The PAKE algorithm to write: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.

Returns: `void`

Description

This function overwrites any PAKE algorithm previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_get_primitive` (function)

Retrieve the primitive from a PAKE cipher suite.

```
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_pake_primitive_t`

The primitive stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_primitive` (function)

Declare the primitive for a PAKE cipher suite.

```
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
```

Parameters

`cipher_suite` The cipher suite object to write to.

`primitive` The PAKE primitive to write: a value of type `psa_pake_primitive_t`. If this is `0`, the primitive type in `cipher_suite` becomes unspecified.

Returns: `void`

Description

This function overwrites any primitive previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_get_hash` (function)

Retrieve the hash algorithm from a PAKE cipher suite.

```
psa_pake_primitive_t psa_pake_cs_get_hash(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite` The cipher suite object to query.

Returns: `psa_pake_primitive_t`

The hash algorithm stored in the cipher suite object. The return value is `PSA_ALG_NONE` if the PAKE is not parametrized by a hash algorithm, or if the hash algorithm is not set.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_hash` (function)

Declare the hash algorithm for a PAKE cipher suite.

```
void psa_pake_cs_set_hash(psa_pake_cipher_suite_t* cipher_suite,
                          psa_algorithm_t hash_alg);
```

Parameters

`cipher_suite` The cipher suite object to write to.

`hash_alg` The hash algorithm to write: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. If this is `PSA_ALG_NONE`, the hash algorithm in `cipher_suite` becomes unspecified.

Returns: `void`

Description

This function overwrites any hash algorithm previously set in `cipher_suite`.

The documentation of individual PAKE algorithms specifies which hash algorithms are compatible, or if no hash algorithm is required.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

2.2.4 PAKE roles

Some PAKE algorithms need to know which role each participant is taking in the algorithm. For example:

- Augmented PAKE algorithms typically have a client and a server participant.
- Some symmetric PAKE algorithms need to assign an order to the participants.

`psa_pake_role_t` (type)

Encoding of the application role in a PAKE algorithm.

```
typedef uint8_t psa_pake_role_t;
```

This type is used to encode the application's role in the algorithm being executed. For more information see the documentation of individual PAKE role constants.

`PSA_PAKE_ROLE_NONE` (macro)

A value to indicate no role in a PAKE algorithm.

```
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
```

This value can be used in a call to `psa_pake_set_role()` for symmetric PAKE algorithms which do not assign roles.

`PSA_PAKE_ROLE_FIRST` (macro)

The first peer in a balanced PAKE.

```
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call `psa_pake_set_role()`, or use `PSA_PAKE_ROLE_NONE` as the role parameter.

PSA_PAKE_ROLE_SECOND (macro)

The second peer in a balanced PAKE.

```
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call `psa_pake_set_role()`, or use `PSA_PAKE_ROLE_NONE` as the role parameter.

PSA_PAKE_ROLE_CLIENT (macro)

The client in an augmented PAKE.

```
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
```

Augmented PAKE algorithms need to differentiate between client and server.

PSA_PAKE_ROLE_SERVER (macro)

The server in an augmented PAKE.

```
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
```

Augmented PAKE algorithms need to differentiate between client and server.

2.2.5 PAKE step types

psa_pake_step_t (type)

Encoding of input and output steps for a PAKE algorithm.

```
typedef uint8_t psa_pake_step_t;
```

Some PAKE algorithms need to exchange more data than a single key share. This type encodes additional input and output steps for such algorithms.

PSA_PAKE_STEP_KEY_SHARE (macro)

The key share being sent to or received from the peer.

```
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
```

The format for both input and output using this step is the same as the format for public keys on the group specified by the PAKE operation's primitive.

The public key formats are defined in the documentation for `psa_export_public_key()`.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

`PSA_PAKE_STEP_ZK_PUBLIC` (macro)

A Schnorr NIZKP public key.

```
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
```

This is the ephemeral public key in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *V* in [RFC8235].

The format for both input and output at this step is the same as that for public keys on the group specified by the PAKE operation's primitive.

For more information on the format, consult the documentation of `psa_export_public_key()`.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

`PSA_PAKE_STEP_ZK_PROOF` (macro)

A Schnorr NIZKP proof.

```
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
```

This is the proof in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *r* in [RFC8235].

Both for input and output, the value at this step is an integer less than the order of the group specified by the PAKE operation's primitive. The format depends on the group as well:

- For Montgomery curves, the encoding is little endian.
- For other Elliptic curves, and for Diffie-Hellman groups, the encoding is big endian. See [SEC1] §2.3.8.

In both cases leading zeroes are allowed as long as the length in bytes does not exceed the byte length of the group order.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

2.2.6 Multi-part PAKE operations

`psa_pake_operation_t` (type)

The type of the state object for PAKE operations.

```
typedef /* implementation-defined type */ psa_pake_operation_t;
```

Before calling any function on a PAKE operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

```
psa_pake_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

```
static psa_pake_operation_t operation;
```

- Initialize the object to the initializer `PSA_PAKE_OPERATION_INIT`, for example:

```
psa_pake_operation_t operation = PSA_PAKE_OPERATION_INIT;
```

- Assign the result of the function `psa_pake_cipher_suite_init()` to the object, for example:

```
psa_pake_operation_t operation;
operation = psa_pake_operation_init();
```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

PSA_PAKE_OPERATION_INIT (macro)

This macro returns a suitable initializer for a PAKE operation object of type `psa_pake_operation_t`.

```
#define PSA_PAKE_OPERATION_INIT /* implementation-defined value */
```

psa_pake_operation_init (function)

Return an initial value for a PAKE operation object.

```
psa_pake_operation_t psa_pake_operation_init(void);
```

Returns: `psa_pake_operation_t`

psa_pake_setup (function)

Set the session information for a password-authenticated key exchange.

```
psa_status_t psa_pake_setup(psa_pake_operation_t *operation,
                            const psa_pake_cipher_suite_t *cipher_suite);
```

Parameters

`operation`
The operation object to set up. It must have been initialized as per the documentation for `psa_pake_operation_t` and not yet in use.

`cipher_suite`
The cipher suite to use. A PAKE cipher suite fully characterizes a PAKE algorithm, including the PAKE algorithm.

Returns: `psa_status_t`

`PSA_SUCCESS`
Success.

`PSA_ERROR_BAD_STATE`
The following conditions can result in this error:

- The operation state is not valid: it must be inactive.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT`
The following conditions can result in this error:

- The algorithm in `cipher_suite` is not a PAKE algorithm.
- The PAKE primitive in `cipher_suite` is not compatible with the PAKE algorithm.
- The hash algorithm in `cipher_suite` is invalid, or not compatible with the PAKE algorithm and primitive.

`PSA_ERROR_NOT_SUPPORTED`
The following conditions can result in this error:

- The algorithm in `cipher_suite` is not a supported PAKE algorithm.
- The PAKE primitive in `cipher_suite` is not supported or not compatible with the PAKE algorithm.
- The hash algorithm in `cipher_suite` is not supported, or not compatible with the PAKE algorithm and primitive.

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

Description

The sequence of operations to set up a password-authenticated key exchange operation is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for `psa_pake_operation_t`. For example, using `PSA_PAKE_OPERATION_INIT`.
3. Call `psa_pake_setup()` to specify the cipher suite.
4. Call `psa_pake_set_xxx()` functions on the operation to complete the setup. The exact sequence of `psa_pake_set_xxx()` functions that needs to be called depends on the algorithm in use.

A typical sequence of calls to perform a password-authenticated key exchange:

1. Call `psa_pake_output(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to get the key share that needs to be sent to the peer.
2. Call `psa_pake_input(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to provide the key share that was received from the peer.
3. Depending on the algorithm additional calls to `psa_pake_output()` and `psa_pake_input()` might be necessary.
4. Call `psa_pake_get_implicit_key()` for accessing the shared secret.

Refer to the documentation of individual PAKE algorithms for details on the required set up and operation for each algorithm. See *PAKE algorithms* on page 12.

If an error occurs at any step after a call to `psa_pake_setup()`, the operation will need to be reset by a call to `psa_pake_abort()`. The application may call `psa_pake_abort()` at any time after the operation has been initialized.

After a successful call to psa_pake_setup(), the application must eventually terminate the operation. The following events terminate an operation:

- A call to psa_pake_abort().
- A successful call to psa_pake_get_implicit_key().

psa_pake_set_password_key (function)

Set the password for a password-authenticated key exchange using a key.

```
psa_status_t psa_pake_set_password_key(psa_pake_operation_t *operation,
                                       psa_key_id_t password);
```

Parameters

operation — Active PAKE operation.

password — Identifier of the key holding the password or a value derived from the password. It must remain valid until the operation terminates. It must be of type PSA_KEY_TYPE_PASSWORD or PSA_KEY_TYPE_PASSWORD_HASH. It must allow the usage PSA_KEY_USAGE_DERIVE.

Returns: psa_status_t

PSA_SUCCESS — Success.

PSA_ERROR_BAD_STATE — The following conditions can result in this error:

- The operation state is not valid: it must be active, and psa_pake_set_password_key(), psa_pake_input(), and psa_pake_output() must not have been called yet.
- The library requires initializing by a call to psa_crypto_init().

PSA_ERROR_INVALID_HANDLE — password is not a valid key identifier.

PSA_ERROR_NOT_PERMITTED — The key does not have the PSA_KEY_USAGE_DERIVE flag, or it does not permit the operation's algorithm.

PSA_ERROR_INVALID_ARGUMENT — The following conditions can result in this error:

- The key type for password is not PSA_KEY_TYPE_PASSWORD or PSA_KEY_TYPE_PASSWORD_HASH.
- password is not compatible with the operation's cipher suite.

PSA_ERROR_NOT_SUPPORTED — The key type or key size of password is not supported with the operation's cipher suite.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

Description

Refer to the documentation of individual PAKE algorithms for constraints on the format and content of valid passwords. See *PAKE algorithms* on page 12.

psa_pake_set_user (function)

Set the user ID for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_user(psa_pake_operation_t *operation,
                               const uint8_t *user_id,
                               size_t user_id_len);
```

Parameters

`operation`	Active PAKE operation.
`user_id`	The user ID to authenticate with.
`user_id_len`	Size of the `user_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_user()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`user_id` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `user_id` is not supported by the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function to set the user ID. For PAKE algorithms that associate a user identifier with both participants in the session, also call `psa_pake_set_peer()` with the peer ID. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

psa_pake_set_peer (function)

Set the peer ID for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_peer(psa_pake_operation_t *operation,
                               const uint8_t *peer_id,
                               size_t peer_id_len);
```

Parameters

`operation` Active PAKE operation.

`peer_id` The peer's ID to authenticate.

`peer_id_len` Size of the `peer_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS` Success.

`PSA_ERROR_BAD_STATE` The following conditions can result in this error:

- The operation state is not valid: it must be active, and `psa_pake_set_peer()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet.
- Calling `psa_pake_set_peer()` is invalid with the operation's algorithm.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT` `peer_id` is not valid for the operation's algorithm and cipher suite.

`PSA_ERROR_NOT_SUPPORTED` The value of `peer_id` is not supported by the implementation.

`PSA_ERROR_NOT_SUPPORTED`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

Description

Call this function in addition to `psa_pake_set_user()` for PAKE algorithms that associate a user identifier with both participants in the session. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

`psa_pake_set_role` (function)

Set the application role for a password-authenticated key exchange.

```
psa_status_t psa_pake_set_role(psa_pake_operation_t *operation,
                               psa_pake_role_t role);
```

Parameters

`operation` Active PAKE operation.

`role` A value of type `psa_pake_role_t` indicating the application role in the PAKE algorithm. See *PAKE roles* on page 23.

Returns: `psa_status_t`

`PSA_SUCCESS` Success.

`PSA_ERROR_BAD_STATE` The following conditions can result in this error:

- The operation state is not valid: it must be active, and `psa_pake_set_role()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT`
`role` is not a valid PAKE role in the operation's algorithm.

`PSA_ERROR_NOT_SUPPORTED`
`role` is not a valid PAKE role, or is not supported for the operation's algorithm.

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

Description

Not all PAKE algorithms need to differentiate the communicating participants. For PAKE algorithms that do not require a role to be specified, the application can do either of the following:

- Not call `psa_pake_set_role()` on the PAKE operation.
- Call `psa_pake_set_role()` with the `PSA_PAKE_ROLE_NONE` role.

Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

`psa_pake_output` (function)

Get output for a step of a password-authenticated key exchange.

```
psa_status_t psa_pake_output(psa_pake_operation_t *operation,
                             psa_pake_step_t step,
                             uint8_t *output,
                             size_t output_size,
                             size_t *output_length);
```

Parameters

`operation`
Active PAKE operation.

`step`
The step of the algorithm for which the output is requested.

`output`
Buffer where the output is to be written. The format of the output depends on the `step`, see *PAKE step types* on page 24.

`output_size`
Size of the `output` buffer in bytes. This must be appropriate for the cipher suite and output step:

- A sufficient output size is `PSA_PAKE_OUTPUT_SIZE(alg, primitive, step)` where `alg` and `primitive` are the PAKE algorithm and primitive in the operation's cipher suite, and `step` is the output step.
- `PSA_PAKE_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported PAKE algorithm, primitive and step.

`output_length`
On success, the number of bytes of the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`
: Success. The first `(*output_length)` bytes of `output` contain the output.

`PSA_ERROR_BAD_STATE`
: The following conditions can result in this error:

- The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_BUFFER_TOO_SMALL`
: The size of the `output` buffer is too small. `PSA_PAKE_OUTPUT_SIZE()` or `PSA_PAKE_OUTPUT_MAX_SIZE` can be used to determine a sufficient buffer size.

`PSA_ERROR_INVALID_ARGUMENT`
: `step` is not compatible with the operation's algorithm.

`PSA_ERROR_NOT_SUPPORTED`
: `step` is not supported with the operation's algorithm.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_input` (function)

Provide input for a step of a password-authenticated key exchange.

```
psa_status_t psa_pake_input(psa_pake_operation_t *operation,
                            psa_pake_step_t step,
                            const uint8_t *input,
                            size_t input_length);
```

Parameters

`operation`
: Active PAKE operation.

`step`
: The step for which the input is provided.

`input`
Buffer containing the input. The format of the input depends on the `step`, see *PAKE step types* on page 24.

`input_length`
Size of the `input` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`
Success.

`PSA_ERROR_BAD_STATE`
The following conditions can result in this error:

- The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_SIGNATURE`
The verification fails for a `PSA_PAKE_STEP_ZK_PROOF` input step.

`PSA_ERROR_INVALID_ARGUMENT`
The following conditions can result in this error:

- `step` is not compatible with the operation's algorithm.
- The input is not valid for the operation's algorithm, cipher suite or `step`.

`PSA_ERROR_NOT_SUPPORTED`
The following conditions can result in this error:

- `step` is not supported with the operation's algorithm.
- The input is not supported for the operation's algorithm, cipher suite or `step`.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

`PSA_PAKE_INPUT_SIZE()` or `PSA_PAKE_INPUT_MAX_SIZE` can be used to allocate buffers of sufficient size to transfer inputs that are received from the peer into the operation.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_get_implicit_key` (function)

Pass the implicitly confirmed shared secret from a PAKE into a key derivation operation.

```
psa_status_t psa_pake_get_implicit_key(psa_pake_operation_t *operation,
                                       psa_key_derivation_operation_t *output);
```

Parameters

`operation` Active PAKE operation.

`output` A key derivation operation that is ready for an input step of type `PSA_KEY_DERIVATION_INPUT_SECRET`.

Returns: `psa_status_t`

`PSA_SUCCESS` Success. Use the `output` key derivation operation to continue with derivation of keys or data.

`PSA_ERROR_BAD_STATE` The following conditions can result in this error:

- The state of PAKE operation `operation` is not valid: it must be active, with all setup, input, and output steps complete.
- The state of key derivation operation `output` is not valid for the `PSA_KEY_DERIVATION_INPUT_SECRET` step.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT` `PSA_KEY_DERIVATION_INPUT_SECRET` is not compatible with the algorithm in the `output` key derivation operation.

`PSA_ERROR_NOT_SUPPORTED` Input from a PAKE is not supported by the algorithm in the `output` key derivation operation.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

At this step in the PAKE operation there is a cryptographic guarantee that only an authenticated participant who used the same password is able to compute the key. But there is no guarantee that the peer is the participant it claims to be, and was able to compute the same key.

In this situation, the authentication is only implicit. Since the peer is not authenticated, no action should be taken that assumes that the peer is who it claims to be For example, do not access restricted files on the peer's behalf until an explicit authentication has succeeded.

This function can be called after the key exchange phase of the operation has completed. It injects the shared secret output of the PAKE into the provided key derivation operation. The input step `PSA_KEY_DERIVATION_INPUT_SECRET` is used to input the shared key material into the key derivation operation.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information. See *PAKE algorithms* on page 12.

When this function returns successfully, `operation` becomes inactive. If this function returns an error status, both the `operation` and the `key_derivation` operations enter an error state and must be aborted by calling `psa_pake_abort()` and `psa_key_derivation_abort()` respectively.

`psa_pake_abort` (function)

Abort a PAKE operation.

```
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
```

Parameters

`operation`	Initialized PAKE operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Aborting an operation frees all associated resources except for the `operation` object itself. Once aborted, the operation object can be reused for another operation by calling `psa_pake_setup()` again.

This function can be called any time after the operation object has been initialized as described in `psa_pake_operation_t`.

In particular, calling `psa_pake_abort()` after the operation has been terminated by a call to `psa_pake_abort()` or `psa_pake_get_implicit_key()` is safe and has no effect.

2.2.7 Support macros

`PSA_PAKE_OUTPUT_SIZE` (macro)

Sufficient output buffer size for `psa_pake_output()`, in bytes.

```
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
```

Parameters

`alg`	A PAKE algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.
`primitive`	A primitive of type `psa_pake_primitive_t` that is compatible with algorithm `alg`.
`output_step`	A value of type `psa_pake_step_t` that is valid for the algorithm `alg`.

Returns

A sufficient output buffer size for the specified PAKE algorithm, primitive, and output step. An implementation can return either `0` or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_pake_output()` will not fail due to an insufficient buffer size. The actual size of the output might be smaller in any given call.

See also `PSA_PAKE_OUTPUT_MAX_SIZE`

PSA_PAKE_OUTPUT_MAX_SIZE (macro)

Sufficient output buffer size for `psa_pake_output()` for any of the supported PAKE algorithms, primitives and output steps.

```
#define PSA_PAKE_OUTPUT_MAX_SIZE /* implementation-defined value */
```

If the size of the output buffer is at least this large, it is guaranteed that `psa_pake_output()` will not fail due to an insufficient buffer size.

See also `PSA_PAKE_OUTPUT_SIZE()`.

PSA_PAKE_INPUT_SIZE (macro)

Sufficient buffer size for inputs to `psa_pake_input()`.

```
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
```

Parameters

`alg` A PAKE algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.

`primitive` A primitive of type `psa_pake_primitive_t` that is compatible with algorithm `alg`.

`input_step` A value of type `psa_pake_step_t` that is valid for the algorithm `alg`.

Returns

A sufficient buffer size for the specified PAKE algorithm, primitive, and input step. An implementation can return either `0` or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

The value returned by this macro is guaranteed to be large enough for any valid input to `psa_pake_input()` in an operation with the specified parameters.

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also `PSA_PAKE_INPUT_MAX_SIZE`

PSA_PAKE_INPUT_MAX_SIZE (macro)

Sufficient buffer size for inputs to `psa_pake_input()` for any of the supported PAKE algorithms, primitives and input steps.

```
#define PSA_PAKE_INPUT_MAX_SIZE /* implementation-defined value */
```

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also `PSA_PAKE_INPUT_SIZE()`.

Appendix A: Example header file

The API elements in this specification, once finalized, will be defined in `psa/crypto.h`.

This is an example of the header file definition of the PAKE API elements. This can be used as a starting point or reference for an implementation.

> **Note:**
>
> Not all of the API elements are fully defined. An implementation must provide the full definition.
>
> The header will not compile without these missing definitions, and might require reordering to satisfy C compilation rules.

psa/crypto.h

```
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
typedef uint8_t psa_pake_family_t;
typedef /* implementation-defined type */ psa_pake_operation_t;
typedef uint32_t psa_pake_primitive_t;
typedef uint8_t psa_pake_primitive_type_t;
typedef uint8_t psa_pake_role_t;
typedef uint8_t psa_pake_step_t;
#define PSA_ALG_IS_PAKE(alg) /* specification-defined value */
#define PSA_ALG_JPAKE ((psa_algorithm_t)0x0a000100)
#define PSA_PAKE_CIPHER_SUITE_INIT /* implementation-defined value */
#define PSA_PAKE_INPUT_MAX_SIZE /* implementation-defined value */
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
#define PSA_PAKE_OPERATION_INIT /* implementation-defined value */
#define PSA_PAKE_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
```

(continues on next page)

(continued from previous page)

```
psa_pake_primitive_t psa_pake_cs_get_hash(const psa_pake_cipher_suite_t* cipher_suite);
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
void psa_pake_cs_set_hash(psa_pake_cipher_suite_t* cipher_suite,
                          psa_algorithm_t hash_alg);
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
psa_status_t psa_pake_get_implicit_key(psa_pake_operation_t *operation,
                                       psa_key_derivation_operation_t *output);
psa_status_t psa_pake_input(psa_pake_operation_t *operation,
                            psa_pake_step_t step,
                            const uint8_t *input,
                            size_t input_length);
psa_pake_operation_t psa_pake_operation_init(void);
psa_status_t psa_pake_output(psa_pake_operation_t *operation,
                             psa_pake_step_t step,
                             uint8_t *output,
                             size_t output_size,
                             size_t *output_length);
psa_status_t psa_pake_set_password_key(psa_pake_operation_t *operation,
                                       psa_key_id_t password);
psa_status_t psa_pake_set_peer(psa_pake_operation_t *operation,
                               const uint8_t *peer_id,
                               size_t peer_id_len);
psa_status_t psa_pake_set_role(psa_pake_operation_t *operation,
                               psa_pake_role_t role);
psa_status_t psa_pake_set_user(psa_pake_operation_t *operation,
                               const uint8_t *user_id,
                               size_t user_id_len);
psa_status_t psa_pake_setup(psa_pake_operation_t *operation,
                            const psa_pake_cipher_suite_t *cipher_suite);
```

Appendix B: Example macro implementations

This section provides example implementations of the function-like macros that have specification-defined values.

Note:

In a future version of this specification, these example implementations will be replaced with a pseudo-code representation of the macro's computation in the macro description.

The examples here provide correct results for the valid inputs defined by each API, for an implementation that supports all of the defined algorithms and key types. An implementation can provide alternative definitions of these macros:

```
#define PSA_ALG_IS_PAKE(alg) \
    (((alg) & 0x7f000000) == 0x0a000000)

#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    ((pake_bits & 0xFFFF) != pake_bits) ? 0 :                 \
    ((psa_pake_primitive_t) (((pake_type) << 24 |             \
            (pake_family) << 16) | (pake_bits)))
```

Index of API elements