Security Middleware Library - SMW

Release 2.5

NXP

Feb 02, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The Security Middleware library provides an application's generic API to perform Security operations supported by the Secure Subsystems present in the system.

This documentation is automatically generated from the source code.

It describes the public functions and the public structures used to perform the Security operations.

It also describes the return codes of these public functions.

CHAPTER

TWO

HOW TO WRITE A CONFIGURATION FILE

This section describes how to write a configuration file. It gives the syntactic rules, the list of valid inputs and some examples.

2.1 Syntactic rules

The configuration format must respect following rules:

- Characters must be encoded in ASCII.
- Semicolon specifies end of line.
- Colon is a separator of multiples entries.
- Spaces and line separators are ignored.
- Decimal numbers are written using the US/UK format (i.e. separator is '.').
- Negative numbers are preceded by '-'.
- String must not be quoted.
- Commented sections start with '/*' and finish with '*/'. Comments are ignored.
- The first tag to define is the **VERSION** tag specifying the parser version compatibility.
- The tag **PSA_DEFAULT**, if present, must be after the tag **VERSION**. If present after the first occurrence of **[SECURE_SUBSYSTEM]**, it is ignored. The possible values are the Secure Subsystems names listed in *List of Secure Subsystems*. Adding option **ALT** (":ALT") after the Secure Subsystem name allows the selection of another Secure Subsystem if the default one doesn't support the requested Security Operation.
- There must be at least one occurrence of **[SECURE_SUBSYSTEM]**. This tag is the starter of a Secure Subsystem configuration.
- There must be only one block defining a Secure Subsystem.
- There must be only one **<string: name of subsystem>** per **[SECURE_SUBSYSTEM]**.
- The load/unload method **<string: load/unload method>** is optional. Only one occurrence is allowed if present.
- There must be at least one occurrence of **[SECURITY_OPERATION]** per **[SECURE_SUBSYSTEM]**.

- There must be one **<string: name of operation>** set in **[SECURITY_OPERATION]**. The possible values of string correspond to the external interfaces of each module as listed in *List of Security Operations*.
- There must be only one block defining a Security Operation for a given Secure Subsystem.
- The Security Operation can define its capabilities values (e.g. key types, hash algorithms…) using tags **<param#>_VALUES** (as listed in *List of Security Operation values tag*). Each value is a non-quoted string separated by a colon.
- The Security Operation can define its capabilities range using tags **<param#>_RANGE** (as listed in *List of Security Operation range tag*). Range values are integer defining minimum and/or maximum capability value.

Configuration file subsystem/operation definition pair represents the subsystem selection order when Secure Subsystem is not specified in the operation arguments. Note: Secure Subsystem is implicit when an operation uses a key identifier of a key already present in the Secure Subsystem key storage.

2.2 Secure Subsystem definition

```
[SECURE_SUBSYSTEM]
    <string: name of subsystem>;
    <string: load/unload method>;
    [SECURITY_OPERATION]
    ...
    [SECURITY_OPERATION]
    ...
```

The Secure Subsystems definition starts with the tag [SECURE_SUBSYSTEM] and is followed by its string name. The table below lists all Secure Subsystems supported by the Security Middleware library.

List of Secure Subsystems:

Secure Subsystem string name	Description
HSM	Use the HSM/SECO protected secure mode on certain i.MX8 device.
TEE	Use the Secure OS called OPTEE and running in ARM Trustzone Secure world.
ELE	**Use the ELE (EdgeLock Enclave) protected secure mode on:** • i.MX8ULP • i.MX9x

A different load and unload method can be specified for each Secure Subsystem thru the <string: load/unload method> string following the subsystem's string name. The following table defines the possible string value of the load/unload method.

List of Secure Subsystem load/unload methods:

Load/Unload string method	Description
AT_FIRST_CALL_LOAD	At first Secure Subsystem call, the Secure Subsystem is loaded. The Secure Subsystem is unloaded when the configuration is unloaded.
AT_CONTEXT_CREATION_DESTRUCTION	At Secure Subsystem context creation, the Secure Subsystem is loaded. It is unloaded when the Secure Subsystem context is destroyed.

Following the Secure Subsystem capabilities, the configuration contains one or more operations associated to the Secure Subsystem as defined Security Operation definition.

2.3 Security Operation definition

```
[SECURITY_OPERATION]
    <string: name of operation>;
    /* A combination of the lines below describes */
    /* the Secure Subsystem capabilities for this Security Operation. */
    <param1>_VALUES=<value1>:<value2>:<value3>;
    <param2>_RANGE=<integer: min>:<integer: max>;
    <param3>_RANGE=:<integer: max>; /* threshold lower than */
    <param4>_RANGE=<integer: min>:; /* threshold greater than */
```

The Security Operation starts with the tag [SECURITY_OPERATION] and is followed by its string name. The table below lists all Security Operations supported by the Security Middleware library.

List of Security Operations:

Security Operation string name	Description
GENERATE_KEY	Generate a cryptographic key (private, keypair). Public key can be exported.
DERIVE_KEY	Derive a key from an existing cryptographic key.
UPDATE_KEY	Update imported or generated key attributes.
IMPORT_KEY	Import cryptographic key (public, private, keypair).
EXPORT_KEY	Export cryptographic key. Private key exportation is function of the Secure Subsystem capabilities.
DELETE_KEY	Delete an imported or generated cryptographic key.
CANCEL_OPERATION	Cancel an active operation context.
COPY_CONTEXT	Copy an active operation context.
HASH	Hash a message.
HMAC	Keyed-hash authentication of a message. **Deprecated use MAC**
MAC	Message Authentication Code.
SIGN	Sign a message.
VERIFY	Verify the signature of a message.
CIPHER	Cipher encryption and decryption.
CIPHER_MULTI_PART	Cipher multi-part encryption and decryption.
AUTHENTICATE_ENCRYPT	Encrypt and sign a message.
AUTHENTICATE_DECRYPT	Decrypt and verify a message.
RNG	Generate a Random data number.
DEVICE_LIFECYCLE	Get and Set device lifecycle.
STORAGE_STORE	Store data in secure storage.
STORAGE_RETRIEVE	Retrieve data from secure storage.
STORAGE_DELETE	Delete data from secure storage.

Each Security Operations definition can specify capabilities using Values and Range tags definition as listed in the following tables.

List of Security Operation values tag:

Tag Values	Description
ALGO_VALUES	Define the operation algorithms supported.
MODE_VALUES	Define the modes supported for the operation algorithms.
HASH_ALGO_VALUES	Define the Hash operation algorithms supported for the operation.
MAC_ALGO_VALUES	Define the MAC operation algorithms supported for the operation.
KEY_TYPE_VALUES	Define the Key types supported for the operation.
SIGN_TYPE_VALUES	Define the signature types supported for signature operations (sign and verify).
OP_TYPE_VALUES	Define the type of operation when it has multiple possibilities (ex: encryption vs decryption for cipher operation).

List of Security Operation range tag:

Tag Range	Description
<KEY_TYPE>_SIZE_RANGE	Define the minimum and maximum key size bits of a key type listed by the **KEY_TYPE_VALUES** tag.
RNG_LENGTH_RANGE	Define the length range of a random number generated with the RNG operation.

Notice that all Values or Range are not useful for each operation. Refer to each operation to get the tags that could be defined and the corresponding value.

2.4 Example

On Linux the plaintext configuration may be a text file. This example defines the configuration supporting 2 Secure Subsystems: OPTEE and HSM.

PSA default Secure Subsystem is OPTEE. Secure Subsystem selection is enabled if OPTEE does not support the requested Security Operation.

OPTEE configuration:

- Subsystem is loaded/unloaded when configuration is loaded and unloaded, refer to Secure Subsystems definition.
- Cipher AES (ECB and CBC) and DES (ECB and CBC) operation. OPTEE is the default subsystem for this operation for the defined keys and modes.
- All keys defined by the Security Middleware can be generated using OPTEE Secure Subsystem.

HSM configuration:

- Subsystem is loaded/unloaded with the default method as defined in Secure Subsystems definition.
- Digest SHA256 operation.

- Generate 128 bits to 256 bits AES keys.
- Generate 56 bits DES keys.
- HSM is the default subsystem for this operation for the defined key capabilities.

```
/* Configuration file */
VERSION=1;
PSA_DEFAULT=TEE:ALT;
[SECURE_SUBSYSTEM]
    TEE;
    /* Load/unload method */
    AT_FIRST_CALL_LOAD;
    [SECURITY_OPERATION]
        CIPHER;
        /* Only AES and DES keys are supported */
        KEY_TYPE_VALUES=AES:DES;
        /* Only ECB and CBC modes are supported */
        MODE_VALUES=ECB:CBC;
    [SECURITY_OPERATION]
        GENERATE_KEY;
        /* No specific capabilities - all parameters are accepted */
[SECURE_SUBSYSTEM]
    HSM;
    /* No Load/unload method specified. Default is 1. */
    [SECURITY_OPERATION]
        HASH;
        HASH_ALGO_VALUES=SHA256;
    [SECURITY_OPERATION]
        GENERATE_KEY;
        /* Only AES and DES algorithms are supported */
        KEY_TYPE_VALUES=AES:DES;
        /* AES key size allowed is between 128 bits and 256 bits */
        AES_SIZE_RANGE=128:256;
        /* DES key size allowed is 56 bits */
        DES_SIZE_RANGE=56:56;
```

CHAPTER

THREE

PUBLIC APIS

Two sets of APIs are exposed:

- The NXP's Security Middleware APIs called SMW APIs
- The ARM's Platform Security Architecture APIs called PSA APIs

Both APIs set are working independently without concurrence.

3.1 SMW APIs

3.1.1 Library information APIs

3.1.1.1 Introduction

Library user can get general library information using following APIs.

3.1.1.2 smw_get_version

enum *smw_status_code* **smw_get_version**(unsigned int *major, unsigned int *minor)

Get the library version.

Parameters

- **major** (unsigned int*) – Library major version.
- **minor** (unsigned int*) – Library minor version.

3.1.1.2.1 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  Success
- **SMW_STATUS_INVALID_PARAM:**
  Either major or minor parameter is NULL

3.1.2 Configuration APIs

3.1.2.1 Introduction

The configuration APIs allow user of the library to:

- Get information about the state of the Secure Subsystems.
- Get the capabilities of the library operations.
- Load/Unload library configuration.

3.1.2.2 smw_config_subsystem_present

enum *smw_status_code* **smw_config_subsystem_present**(*smw_subsystem_t* subsystem)

Check if the subsystem is present or not.

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem.

3.1.2.2.1 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  `subsystem` is present
- **SMW_STATUS_INVALID_PARAM:**
  `subsystem` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `subsystem` is not a valid string

3.1.2.3 smw_config_subsystem_loaded

enum *smw_status_code* **smw_config_subsystem_loaded**(*smw_subsystem_t* subsystem)

Return if the subsystem is loaded or not.

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem.

3.1.2.3.1 Return

See *enum smw_status_code*

- **SMW_STATUS_SUBSYSTEM_LOADED:**
  `subsystem` is loaded
- **SMW_STATUS_SUBSYSTEM_NOT_LOADED:**
  `subsystem` is not loaded

- **SMW_STATUS_INVALID_PARAM:**
  `subsystem` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `subsystem` is not a valid string
- **SMW_STATUS_INVALID_LIBRARY_CONTEXT:**
  Library context is not valid

3.1.2.4 smw_config_check_digest

enum *smw_status_code* **smw_config_check_digest**(*smw_subsystem_t* subsystem, *smw_hash_algo_t* algo)

Check if a digest `algo` is supported

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem (if NULL default subsystem).
- **algo** (*smw_hash_algo_t*) – Digest algorithm name.

3.1.2.4.1 Description

Function checks if the digest `algo` is supported on the given `subsystem`. If `subsystem` is NULL, default subsystem digest capability is checked.

3.1.2.4.2 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  `algo` is supported
- **SMW_STATUS_INVALID_PARAM:**
  `algo` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `algo` is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  `algo` is not supported

3.1.2.5 struct smw_key_info

struct **smw_key_info**

Key information

3.1.2.5.1 Definition

```
struct smw_key_info {
    smw_key_type_t key_type_name;
    unsigned int security_size;
    unsigned int security_size_min;
    unsigned int security_size_max;
}
```

3.1.2.5.2 Members

key_type_name
Key type name. See *typedef smw_key_type_t*

security_size
Key security size in bits

security_size_min
Key security size minimum in bits

security_size_max
Key security size maximum in bits

3.1.2.6 smw_config_check_generate_key

enum *smw_status_code* **smw_config_check_generate_key**(*smw_subsystem_t* subsystem, struct *smw_key_info* *info)

Check generate key type

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem (if NULL default subsystem).
- **info** (struct *smw_key_info**) – Key information

3.1.2.6.1 Description

Function checks if the key type provided in the `info` structure is supported on the given subsystem.

If `info`‘s security size field is equal 0, returns the security key range size in bits supported by the subsystem for the key type. Else checks if the security size is supported.

If `subsystem` is NULL, default subsystem key generation is checked.

3.1.2.6.2 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  Key type is supported
- **SMW_STATUS_INVALID_PARAM:**
  `info` or `info->key_type_name` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `info->key_type_name` is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  Key type is not supported

3.1.2.7 struct smw_signature_info

struct **smw_signature_info**

Signature operation information

3.1.2.7.1 Definition

```
struct smw_signature_info {
    smw_key_type_t key_type_name;
    smw_hash_algo_t hash_algo;
    smw_signature_type_t signature_type;
}
```

3.1.2.7.2 Members

key_type_name
Key type name. See *typedef smw_key_type_t*

hash_algo
Hash algorithm name. See *typedef smw_hash_algo_t*

signature_type
Signature type name. See *typedef smw_signature_type_t*

3.1.2.8 smw_config_check_sign

enum *smw_status_code* **smw_config_check_sign**(*smw_subsystem_t* subsystem, struct *smw_signature_info* *info)

Check if signature generation operation is supported

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem (if NULL default subsystem).

- **info** (struct *smw_signature_info**) – Signature information

3.1.2.8.1 Description

info key type name field is mandatory. info hash algorithm name and signature type name fields are optional.

Function checks if the key type provided in the info structure is supported on the given subsystem for a signature generation operation. If set, function checks if the hash algorithm is supported on the given subsystem for the signature generation operation. If set, function checks if the signature type is supported on the given subsystem for the signature generation operation.

If subsystem is NULL, default subsystem digest capability is checked.

3.1.2.8.2 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  Signature operation is supported
- **SMW_STATUS_INVALID_PARAM:**
  info or info->key_type_name is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  info->key_type_name is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  Signature operation is not supported

3.1.2.9 smw_config_check_verify

enum *smw_status_code* **smw_config_check_verify**(*smw_subsystem_t* subsystem, struct *smw_signature_info* *info)

Check if signature verification operation is supported

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem (if NULL default subsystem).
- **info** (struct *smw_signature_info**) – Signature information

3.1.2.9.1 Description

info key type name field is mandatory. info hash algorithm name and signature type name fields are optional.

Function checks if the key type provided in the info structure is supported on the given subsystem for a signature verification operation. If set, function checks if the hash algorithm is supported on the given subsystem for the signature verification operation. If set, function checks if the signature type is supported on the given subsystem for the signature verification operation.

If `subsystem` is NULL, default subsystem digest capability is checked.

3.1.2.9.2 Return

See *`enum smw_status_code`*

- **SMW_STATUS_OK:**
  Verify operation is supported
- **SMW_STATUS_INVALID_PARAM:**
  `info` or `info->key_type_name` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `info->key_type_name` is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  Verify operation is not supported

3.1.2.10 struct smw_cipher_info

struct **`smw_cipher_info`**

Cipher operation information

3.1.2.10.1 Definition

```
struct smw_cipher_info {
    bool multipart;
    smw_key_type_t key_type_name;
    smw_cipher_mode_t mode;
    smw_cipher_operation_t op_type;
}
```

3.1.2.10.2 Members

multipart
True if it's a cipher multi-part operation

key_type_name
Key type name. See *`typedef smw_key_type_t`*

mode
Operation mode name. See *`typedef smw_cipher_mode_t`*

op_type
Operation type name. See *`typedef smw_cipher_operation_t`*

3.1.2.11 smw_config_check_cipher

enum *smw_status_code* **smw_config_check_cipher**(*smw_subsystem_t* subsystem, struct *smw_cipher_info* *info)

Check if cipher operation is supported

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem (if NULL default subsystem).
- **info** (struct *smw_cipher_info**) – Cipher information

3.1.2.11.1 Description

Function checks if all fields provided in the `info` structure are supported on the given `subsystem` for a cipher one-shot or multi-part operation.

If `subsystem` is NULL, default subsystem cipher capability is checked.

3.1.2.11.2 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  Cipher operation is supported
- **SMW_STATUS_INVALID_PARAM:**
  `info`, `info->key_type_name`, `info->mode` or `info->op_type` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `info->key_type_name`, `info->mode` or `info->op_type` is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  Cipher operation is not supported

3.1.2.12 struct smw_aead_info

struct **smw_aead_info**

AEAD operation information

3.1.2.12.1 Definition

```
struct smw_aead_info {
    bool multipart;
    smw_key_type_t key_type_name;
    smw_aead_mode_t mode;
    smw_aead_operation_t op_type;
}
```

3.1.2.12.2 Members

multipart
True if it's a AEAD multi-part operation

key_type_name
Key type name. See *typedef smw_key_type_t*

mode
Operation mode name. See *typedef smw_aead_mode_t*

op_type
Operation type name. See *typedef smw_aead_operation_t*

3.1.2.13 smw_config_check_aead

enum *smw_status_code* **smw_config_check_aead**(*smw_subsystem_t* subsystem, struct *smw_aead_info* *info)

Check if AEAD operation is supported

Parameters

- **subsystem** (*smw_subsystem_t*) – Name of the subsystem
- **info** (struct *smw_aead_info**) – AEAD operation information

3.1.2.13.1 Description

Function checks if all fields provided in the `info` structure are supported on the given `subsystem` for a AEAD one-shot or multi-part operation.

If `subsystem` is NULL, default subsystem AEAD capability is checked.

3.1.2.13.2 Return

See *enum smw_status_code*

- **SMW_STATUS_OK:**
  AEAD operation is supported
- **SMW_STATUS_INVALID_PARAM:**
  `info`, `info->key_type_name`, `info->mode` or `info->op_type` is NULL
- **SMW_STATUS_UNKNOWN_NAME:**
  `info->key_type_name`, `info->mode` or `info->op_type` is not a valid string
- **SMW_STATUS_OPERATION_NOT_CONFIGURED:**
  AEAD operation is not supported

3.1.2.14 smw_config_load

enum *smw_status_code* **smw_config_load**(char *buffer, unsigned int size, unsigned int *offset)

Load a configuration.

Parameters

- **buffer** (char*) – pointer to the plaintext configuration.
- **size** (unsigned int) – size of the plaintext configuration.
- **offset** (unsigned int*) – current offset in plaintext configuration.

3.1.2.14.1 Description

This function loads a configuration. The plaintext configuration is parsed and the content is stored in the Configuration database. If the parsing of plaintext configuration fails, `offset` points to the number of characters that have been correctly parsed. The beginning of the remaining plaintext which cannot be parsed is printed out.

3.1.2.14.2 Return

SMW_STATUS_OK - Configuration load is successful SMW_STATUS_INVALID_LIBRARY_CONTEXT - Library context is not valid SMW_STATUS_INVALID_BUFFER - `buffer` is NULL or `size` is 0 SMW_STATUS_CONFIG_ALREADY_LOADED - A configuration is already loaded error code otherwise

3.1.2.15 smw_config_unload

enum *smw_status_code* **smw_config_unload**(void)

Unload the current configuration.

Parameters

- **void** – no arguments

3.1.2.15.1 Description

This function unloads the current configuration. It frees all memory dynamically allocated by SMW.

3.1.2.15.2 Return

SMW_STATUS_OK - Configuration unload is successful SMW_STATUS_INVALID_LIBRARY_CONTEXT - Library context is not valid SMW_STATUS_NO_CONFIG_LOADED - No configuration is loaded

3.1.3 Cryptography APIs

3.1.3.1 struct smw_hash_args

struct **smw_hash_args**

Hash arguments

3.1.3.1.1 Definition

```
struct smw_hash_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    smw_hash_algo_t algo_name;
    unsigned char *input;
    unsigned int input_length;
    unsigned char *output;
    unsigned int output_length;
}
```

3.1.3.1.2 Members

version

Version of this structure

subsystem_name

Secure Subsystem name. See *typedef smw_subsystem_t*

algo_name

Algorithm name. See *typedef smw_hash_algo_t*

input

Location of the stream to be hashed

input_length

Length of the stream to be hashed

output

Location where the digest has to be written

output_length

Length of the digest

3.1.3.1.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.3.2 struct smw_sign_verify_args

struct **smw_sign_verify_args**

Sign or verify arguments

3.1.3.2.1 Definition

```
struct smw_sign_verify_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor *key_descriptor;
    smw_hash_algo_t algo_name;
    unsigned char *message;
    unsigned int message_length;
    unsigned char *signature;
    unsigned int signature_length;
    unsigned char *attributes_list;
    unsigned int attributes_list_length;
}
```

3.1.3.2.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

algo_name
Hash algorithm name. See *typedef smw_hash_algo_t*

message
Location of the message

message_length
Length of the message

signature
Location of the signature

signature_length
Length of the signature

attributes_list
Sign Verify attributes list

attributes_list_length
`attributes_list` length in bytes

3.1.3.2.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.3.3 struct smw_hmac_args

struct **smw_hmac_args**
HMAC arguments

3.1.3.3.1 Definition

```
struct smw_hmac_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor *key_descriptor;
    smw_hash_algo_t algo_name;
    unsigned char *input;
    unsigned int input_length;
    unsigned char *output;
    unsigned int output_length;
}
```

3.1.3.3.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

algo_name
Hash algorithm name. See *typedef smw_hash_algo_t*

input
Location of the stream to be hash-mac'ed

input_length
Length of the stream to be hashed

output
Location where the MAC has to be written

output_length
Length of the MAC

3.1.3.3.3 Description

Deprecated. Will be removed in library version 3.x. Use *smw_mac()* or *smw_mac_verify()*.

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.3.4 struct smw_mac_args

struct **smw_mac_args**
MAC arguments

3.1.3.4.1 Definition

```
struct smw_mac_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor *key_descriptor;
    smw_mac_algo_t algo_name;
    smw_hash_algo_t hash_name;
    unsigned char *input;
    unsigned int input_length;
    unsigned char *mac;
    unsigned int mac_length;
}
```

3.1.3.4.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

algo_name
MAC algorithm name. See *typedef smw_mac_algo_t*

hash_name
Hash algorithm name. See *typedef smw_hash_algo_t*

input
Location of the message to be authenticated

input_length
Length of the message

mac
Location where the MAC has to be written or compared

mac_length
Length of the MAC

3.1.3.4.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used or the Secure Subsystem handling the key specified.

3.1.3.5 struct smw_rng_args

struct **smw_rng_args**
Random number generator arguments

3.1.3.5.1 Definition

```
struct smw_rng_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    unsigned char *output;
    unsigned int output_length;
}
```

3.1.3.5.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

output
Location where the random number has to be written

output_length
Length of the random number

3.1.3.5.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.3.6 struct smw_op_context

struct **smw_op_context**

SMW cryptographic operation context

3.1.3.6.1 Definition

```
struct smw_op_context {
    void *handle;
    void *reserved;
}
```

3.1.3.6.2 Members

handle
Pointer to operation handle

reserved
Reserved data

3.1.3.6.3 Description

Parameters `handle` and `reserved` are set by SMW. They must not be modified by the application

3.1.3.7 struct smw_cipher_init_args

struct **smw_cipher_init_args**

Cipher multi-part initialization arguments

3.1.3.7.1 Definition

```
struct smw_cipher_init_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor **keys_desc;
    unsigned int nb_keys;
    smw_cipher_mode_t mode_name;
    smw_cipher_operation_t operation_name;
    unsigned char *iv;
    unsigned int iv_length;
    struct smw_op_context *context;
}
```

3.1.3.7.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

keys_desc
Pointer to an array of pointers to key descriptors. See *struct smw_key_descriptor*

nb_keys
Number of entries of `keys_desc`

mode_name
Cipher mode name. See *typedef smw_cipher_mode_t*.

operation_name
Cipher operation name. See *typedef smw_cipher_operation_t*

iv
Pointer to initialization vector

iv_length
`iv` length in bytes

context
Pointer to operation context. See *struct smw_op_context*

3.1.3.7.3 Description

Switch `mode`, `iv` is optional and represents:

- Initialization Vector (CBC, CTS)
- Initial Counter Value (CTR)
- Tweak Value (XTS)

3.1.3.8 struct smw_cipher_data_args

struct **smw_cipher_data_args**
Cipher data arguments

3.1.3.8.1 Definition

```
struct smw_cipher_data_args {
    unsigned char version;
    struct smw_op_context *context;
    unsigned char *input;
    unsigned int input_length;
    unsigned char *output;
```

(continues on next page)

(continued from previous page)

```
    unsigned int output_length;
}
```

3.1.3.8.2 Members

version
Version of this structure

context
Pointer to operation context. See *struct smw_op_context*

input
Input data buffer

input_length
`input` length in bytes

output
Output data buffer

output_length
`output` length in bytes

3.1.3.8.3 Description

In case of final operation, `input` and `input_length` are optional.

3.1.3.9 struct smw_cipher_args

struct **smw_cipher_args**
Cipher one-shot arguments

3.1.3.9.1 Definition

```
struct smw_cipher_args {
    struct smw_cipher_init_args init;
    struct smw_cipher_data_args data;
}
```

3.1.3.9.2 Members

init
Initialization arguments. See *struct smw_cipher_init_args*

data
Data arguments. See *struct smw_cipher_data_args*

3.1.3.9.3 Description

Field `context` present in `init` and `data` is ignored.

3.1.3.10 smw_hash

enum *smw_status_code* **smw_hash**(struct *smw_hash_args* *args)

Compute hash.

Parameters

- **args** (struct *smw_hash_args**) – Pointer to the structure that contains the Hash arguments.

3.1.3.10.1 Description

This function computes a hash.

3.1.3.10.2 Return

See ***enum smw_status_code***

- Common return codes

3.1.3.11 smw_sign

enum *smw_status_code* **smw_sign**(struct *smw_sign_verify_args* *args)

Generate a signature.

Parameters

- **args** (struct *smw_sign_verify_args**) – Pointer to the structure that contains the Sign arguments.

3.1.3.11.1 Description

This function generates a signature. When TLS_MAC_FINISH attribute is set, the key type must be TLS_MASTER_KEY.

3.1.3.11.2 Return

enum smw_status_code

- Common return codes
- Specific return codes - Signature

3.1.3.12 smw_verify

enum *smw_status_code* **smw_verify**(struct *smw_sign_verify_args* *args)

Verify a signature.

Parameters

- **args** (struct *smw_sign_verify_args**) – Pointer to the structure that contains the Verify arguments.

3.1.3.12.1 Description

This function verifies a sigature.

3.1.3.12.2 Return

See *enum smw_status_code*

- Common return codes
- Specific return codes - Signature

3.1.3.13 smw_hmac

enum *smw_status_code* **smw_hmac**(struct *smw_hmac_args* *args)

Compute a HASH-MAC.

Parameters

- **args** (struct *smw_hmac_args**) – Pointer to the structure that contains the HMAC arguments.

3.1.3.13.1 Description

Deprecated. Will be removed in library version 3.x. Use *smw_mac()* or *smw_mac_verify()*.

This function computes a Keyed-Hash Message Authentication Code.

3.1.3.13.2 Return

See ***enum smw_status_code***

- Common return codes

3.1.3.14 smw_rng

enum *smw_status_code* **smw_rng**(struct *smw_rng_args* *args)

Compute a random number.

Parameters

- **args** (struct *smw_rng_args**) – Pointer to the structure that contains the RNG arguments.

3.1.3.14.1 Description

This function computes a random number.

3.1.3.14.2 Return

See ***enum smw_status_code***

- Common return codes

3.1.3.15 smw_cipher

enum *smw_status_code* **smw_cipher**(struct *smw_cipher_args* *args)

Cipher one-shot

Parameters

- **args** (struct *smw_cipher_args**) – Pointer to the structure that contains the cipher arguments.

3.1.3.15.1 Description

This function executes a cipher encryption or decryption.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeed, returned error code is SMW_STATUS_OK.

Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

Keys used can be defined as buffer and as key ID. All key types must be identical and must be linked to the same subystem. If at least one key ID is set, subsystem name field of `args` is optional. If set it must be coherent with the key ID.

3.1.3.15.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.3.16 smw_cipher_init

enum *smw_status_code* **`smw_cipher_init`**(struct *smw_cipher_init_args* *args)

Cipher multi-part initialization

Parameters

- **`args`** (`struct` *`smw_cipher_init_args`*`*`) – Pointer to the structure that contains the cipher multi-part initialization arguments.

3.1.3.16.1 Description

This function executes a cipher multi-part encryption or decryption initialization.

Keys used can be defined as buffer and as key ID. All key types must be identical and must be linked to the same subystem. If at least one key ID is set, subsystem name field of `args` is optional. If set it must be coherent with the key ID.

Context structure presents in `args` must be allocated by the application.

3.1.3.16.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.17 smw_cipher_update

enum *smw_status_code* **smw_cipher_update**(struct *smw_cipher_data_args* *args)

Cipher multi-part update

Parameters

- **args** (struct *smw_cipher_data_args**) – Pointer to the structure that contains the cipher multi-part update arguments.

3.1.3.17.1 Description

This function executes a cipher multi-part encryption or decryption update operation.

The context used must be initialized by the cipher multi-part initialization.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeed, returned error code is SMW_STATUS_OK.

Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

If the returned error code is SMW_STATUS_OK, SMW_STATUS_INVALID_PARAM, SMW_STATUS_VERSION_NOT_SUPPORTED or SMW_STATUS_OUTPUT_TOO_SHORT the operation is not terminated and the context remains valid.

3.1.3.17.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.18 smw_cipher_final

enum *smw_status_code* **smw_cipher_final**(struct *smw_cipher_data_args* *args)

Cipher multi-part final

Parameters

- **args** (struct *smw_cipher_data_args**) – Pointer to the structure that contains the cipher multi-part final arguments.

3.1.3.18.1 Description

This function executes a cipher multi-part encryption or decryption final operation.

The context used must be initialized by the cipher multi-part initialization.

Input data field of `args` can be a NULL pointer if no additional data are used.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeed, returned error code is SMW_STATUS_OK and the operation is no terminated (context remains valid) unless required output buffer length is 0.

Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

If the returned error code is SMW_STATUS_INVALID_PARAM, SMW_STATUS_VERSION_NOT_SUPPORTED or SMW_STATUS_OUTPUT_TOO_SHORT the operation is not terminated and the context remains valid.

3.1.3.18.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.3.19 smw_mac

enum *smw_status_code* **`smw_mac`**(struct *smw_mac_args* *args)

Compute a MAC.

Parameters

- **`args`** (`struct` *`smw_mac_args`**) – Pointer to the structure that contains the MAC arguments.

3.1.3.19.1 Description

This function computes a Message Authentication Code.

3.1.3.19.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.3.20 smw_mac_verify

enum *smw_status_code* **smw_mac_verify**(struct *smw_mac_args* *args)

Compute and verify a MAC.

Parameters

- **args** (struct *smw_mac_args**) – Pointer to the structure that contains the MAC arguments.

3.1.3.20.1 Description

This function computes then verifies a Message Authentication Code.

3.1.3.20.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.21 smw_cancel_operation

enum *smw_status_code* **smw_cancel_operation**(struct *smw_op_context* *args)

Cancel on-going cryptographic multi-part operation

Parameters

- **args** (struct *smw_op_context**) – Pointer to operation context.

3.1.3.21.1 Description

If function succeeds, `args` handle field is set to NULL.

3.1.3.21.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.22 smw_copy_context

enum *smw_status_code* **smw_copy_context**(struct *smw_op_context* *dst, struct *smw_op_context* *src)

Copy an operation context

Parameters

- **dst** (struct *smw_op_context**) – Pointer to destination operation context.
- **src** (struct *smw_op_context**) – Pointer to source operation context.

3.1.3.22.1 Description

This function copies an initialized or updated source context to a new created destination context. Parameter `dst` must be allocated by caller.

3.1.3.22.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.3.23 Authentication Encryption/Decryption (AEAD)

3.1.3.23.1 struct smw_aead_init_args

struct **smw_aead_init_args**

AEAD initialization arguments

3.1.3.23.1.1 Definition

```
struct smw_aead_init_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor *key_desc;
    smw_aead_mode_t mode_name;
    smw_aead_operation_t operation_name;
    unsigned char *iv;
    unsigned int iv_length;
    unsigned int aad_length;
    unsigned int tag_length;
    unsigned int plaintext_length;
    struct smw_op_context *context;
}
```

3.1.3.23.1.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *`typedef smw_subsystem_t`*

key_desc
Pointer to a key descriptor object. See *`struct smw_key_descriptor`*

mode_name
AEAD mode name. See *`typedef smw_aead_mode_t`*

operation_name
: AEAD operation name. See *typedef smw_aead_operation_t*

iv
: Pointer to initialization vector

iv_length
: iv length in bytes

aad_length
: Additional authentication data length in bytes

tag_length
: Tag buffer length in bytes

plaintext_length
: Length in bytes of the data to encrypt

context
: Pointer to operation context. See *struct smw_op_context*

3.1.3.23.2 struct smw_aead_data_args

struct **smw_aead_data_args**

AEAD data arguments

3.1.3.23.2.1 Definition

```
struct smw_aead_data_args {
    unsigned char version;
    struct smw_op_context *context;
    unsigned char *input;
    unsigned int input_length;
    unsigned char *output;
    unsigned int output_length;
}
```

3.1.3.23.2.2 Members

version
: Version of this structure

context
: Pointer to operation context. See *struct smw_op_context*

input
: Pointer to input data buffer to be encrypted or decrypted

input_length
: Input data buffer length in bytes

output
: Pointer to output buffer

output_length
Output buffer length in bytes

3.1.3.23.3 struct smw_aead_aad_args

struct **smw_aead_aad_args**
Authentication Encryption AAD arguments

3.1.3.23.3.1 Definition

```
struct smw_aead_aad_args {
    unsigned char version;
    unsigned char *aad;
    unsigned int aad_length;
    struct smw_op_context *context;
}
```

3.1.3.23.3.2 Members

version
Version of this structure

aad
Pointer to additional authentication data

aad_length
AAD length in bytes

context
Pointer to operation context. See *struct smw_op_context*

3.1.3.23.4 struct smw_aead_final_args

struct **smw_aead_final_args**
AEAD final arguments

3.1.3.23.4.1 Definition

```
struct smw_aead_final_args {
    unsigned char version;
    struct smw_aead_data_args data;
    smw_aead_operation_t operation_name;
    unsigned int tag_length;
}
```

3.1.3.23.4.2 Members

version
Version of this structure

data
AEAD data arguments. See *struct smw_aead_data_args*

operation_name
AEAD operation name. See *typedef smw_aead_operation_t*

tag_length
Tag buffer length in bytes

3.1.3.23.5 struct smw_aead_args

struct **smw_aead_args**

AEAD one-shot arguments

3.1.3.23.5.1 Definition

```
struct smw_aead_args {
    struct smw_aead_init_args init;
    struct smw_aead_data_args data;
    unsigned char *aad;
}
```

3.1.3.23.5.2 Members

init
Initialization arguments. See *struct smw_aead_init_args*

data
Data arguments. See *struct smw_aead_data_args*

aad
Pointer to additional authentication data

3.1.3.23.5.3 Description

Field `context` present in `init` and `data` is ignored.

3.1.3.23.6 smw_aead

enum *smw_status_code* **smw_aead**(struct *smw_aead_args* *args)

One-shot AEAD operation.

Parameters

- **args** (struct *smw_aead_args**) – Pointer to the structure that contains the AEAD one-shot arguments.

3.1.3.23.6.1 Description

This function executes one-shot AEAD encryption or decryption operation.

- One-shot AEAD encryption operation:
  - This function encrypts a message and computes the tag.
- One-shot AEAD decryption operation:
  - This function authenticates and decrypts the ciphertext.
  - If the computed tag does not match the supplied tag, the operation will be terminated.
  - The input data field of `args` should be large enough to accommodate the ciphertext and the tag.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeeds, returned error code is SMW_STATUS_OK.

Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation is succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

If output data field of `args` is not a NULL pointer, then

- For encryption operation, output length should be large enough to accommodate both the ciphertext and tag.
- For decryption operation, output length should be large enough to accommodate the plaintext.

3.1.3.23.6.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.23.7 smw_aead_init

enum *smw_status_code* **smw_aead_init**(struct *smw_aead_init_args* *args)

AEAD multi-part initialization.

Parameters

- **args** (struct *smw_aead_init_args**) – Pointer to the structure that contains the AEAD initialization arguments.

3.1.3.23.7.1 Description

This function initializes AEAD multi-part encryption or decryption operation.

Key used can be defined either as a buffer or as a key ID.

3.1.3.23.7.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.23.8 smw_aead_update_add

enum *smw_status_code* **smw_aead_update_add**(struct *smw_aead_aad_args* *args)

Add additional data to the AEAD operation.

Parameters

- **args** (struct *smw_aead_aad_args**) – Pointer to the structure that contains the AEAD additional data arguments.

3.1.3.23.8.1 Description

This function can be called multiple time while the update data (to encrypt or to decrypt) step is not called.

The context used must be initialized by the AEAD multi-part initialization.

3.1.3.23.8.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.23.9 smw_aead_update

enum *smw_status_code* **smw_aead_update**(struct *smw_aead_data_args* *args)

AEAD multi-part update operation

Parameters

- **args** (struct *smw_aead_data_args**) – Pointer to the structure that contains the AEAD multi-part data arguments.

3.1.3.23.9.1 Description

This function executes a AEAD multi-part encryption or decryption update operation.

The context used must be initialized by the AEAD multi-part initialization.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeeds, returned error code is SMW_STATUS_OK.

Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

If the returned error code is SMW_STATUS_OK, SMW_STATUS_INVALID_PARAM, SMW_STATUS_VERSION_NOT_SUPPORTED or SMW_STATUS_OUTPUT_TOO_SHORT the operation is not terminated and the context remains valid.

3.1.3.23.9.2 Return

See *enum smw_status_code*

- Common return codes

3.1.3.23.10 smw_aead_final

enum *smw_status_code* **smw_aead_final**(struct *smw_aead_final_args* *args)

AEAD multi-part encryption/decryption final operation

Parameters

- **args** (struct *smw_aead_final_args**) – Pointer to the structure that contains the AEAD multi-part final arguments.

3.1.3.23.10.1 Description

This function completes the active AEAD multi-part encryption/decryption operation.

The context used must be initialized by the AEAD multi-part initialization.

- AEAD Encryption final operation:
  - This function finishes encrypting a message in an active multi-part AEAD operation and computes the tag.
- AEAD Decryption final operation:
  - This function finishes authenticating and decrypting a message in an active multi-part AEAD operation.
  - If the computed tag does not match the supplied tag, the operation will be terminated. The returned error code is SMW_STATUS_SIGNATURE_INVALID.
  - The input data field of `args` should be large enough to accommodate the ciphertext and the tag.

Output data field of `args` can be a NULL pointer to get the required output buffer length. If this feature succeeds, returned error code is SMW_STATUS_OK. Output length `args` field is updated to the correct value when:

- Output length is bigger than expected. In this case operation succeeded.
- Output length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.

If output data field of `args` is not a NULL pointer, then

- For encryption operation, output length should be large enough to accommodate both the plaintext and tag.
- For decryption operation, output length should be large enough to accommodate the plaintext.

If the returned error code is SMW_STATUS_OK, SMW_STATUS_INVALID_PARAM, SMW_STATUS_VERSION_NOT_SUPPORTED or SMW_STATUS_OUTPUT_TOO_SHORT the operation is not terminated and the context remains valid.

3.1.3.23.10.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.4 Key Manager APIs

3.1.4.1 struct smw_keypair_gen

struct **smw_keypair_gen**

Generic Keypair object

3.1.4.1.1 Definition

```
struct smw_keypair_gen {
    unsigned char *public_data;
    unsigned int public_length;
    unsigned char *private_data;
    unsigned int private_length;
}
```

3.1.4.1.2 Members

public_data

Pointer to the public key

public_length

Length of `public_data` in bytes

private_data

Pointer to the private key

private_length

Length of `private_data` in bytes

3.1.4.2 struct smw_keypair_rsa

struct **smw_keypair_rsa**

RSA Keypair object

3.1.4.2.1 Definition

```
struct smw_keypair_rsa {
    unsigned char *modulus;
    unsigned int modulus_length;
    unsigned char *public_data;
    unsigned int public_length;
    unsigned char *private_data;
    unsigned int private_length;
}
```

3.1.4.2.2 Members

modulus
Pointer to the RSA modulus

modulus_length
Length of `modulus` in bytes

public_data
Pointer to the RSA public exponent

public_length
Length of `public_data` in bytes

private_data
Pointer to the RSA private exponent

private_length
Length of `private_data` in bytes

3.1.4.3 struct smw_keypair_buffer

struct **smw_keypair_buffer**

Keypair buffer

3.1.4.3.1 Definition

```
struct smw_keypair_buffer {
    smw_key_format_t format_name;
    union {
        struct smw_keypair_gen gen;
        struct smw_keypair_rsa rsa;
    } ;
}
```

3.1.4.3.2 Members

format_name
Defines the encoding format of all buffers. See *typedef smw_key_format_t*

{unnamed_union}
anonymous

gen
Generic keypair object definition. See *struct smw_keypair_gen*

rsa
RSA keypair object definition. See *struct smw_keypair_rsa*

3.1.4.3.3 Description

By default if format name is not specified, there will be no encoding (equivalent to "HEX")

3.1.4.4 struct smw_key_descriptor

struct **smw_key_descriptor**

Key descriptor

3.1.4.4.1 Definition

```
struct smw_key_descriptor {
    smw_key_type_t type_name;
    unsigned int security_size;
    unsigned int id;
    struct smw_keypair_buffer *buffer;
}
```

3.1.4.4.2 Members

type_name
Key type name. See *typedef smw_key_type_t*

security_size
Security size in bits

id
Key identifier

buffer
Key pair buffer. See *struct smw_keypair_buffer*

3.1.4.5 struct smw_generate_key_args

struct **smw_generate_key_args**

Key generation arguments

3.1.4.5.1 Definition

```
struct smw_generate_key_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    unsigned char *key_attributes_list;
    unsigned int key_attributes_list_length;
    struct smw_key_descriptor *key_descriptor;
}
```

3.1.4.5.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_attributes_list
Key attributes list. See *typedef smw_attr_key_type_t*

key_attributes_list_length
Length of the Key attributes list

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

3.1.4.5.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default Secure Subsystem configured for this Security Operation is used. The `key_descriptor` fields `type_name` and `security_size` must be given as input to know the type of key to generate. The `key_descriptor` field `buffer` is optional. Only the public key will be returned if the corresponding pointer and size are set. The `key_descriptor` field `id`, if set by the caller (other than 0) will be the created key identifier on operation success. Else the API will returned a new key identifier if `id` is set as 0.

3.1.4.6 struct smw_derive_key_args

struct **smw_derive_key_args**
Key derivation arguments

3.1.4.6.1 Definition

```
struct smw_derive_key_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    smw_kdf_t kdf_name;
    void *kdf_arguments;
    struct smw_key_descriptor *key_descriptor_base;
    unsigned char *key_attributes_list;
    unsigned int key_attributes_list_length;
    struct smw_key_descriptor *key_descriptor_derived;
}
```

3.1.4.6.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

kdf_name
Key derivation function name. See *typedef smw_kdf_t*

kdf_arguments
Key derivation function arguments

key_descriptor_base
Pointer to a Key base descriptor. See *struct smw_key_descriptor*

key_attributes_list
Key attributes list

key_attributes_list_length
Length of the Key attributes list

key_descriptor_derived
Pointer to the Key derived descriptor. See *struct smw_key_descriptor*

3.1.4.6.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default Secure Subsystem configured for this Security Operation is used.

A new key is derived from a given key base (@key_descriptor_base) using the key derivation function `kdf_name`. If the key derivation function requires more arguments, the `kdf_arguments` refers to the associated key derivation function arguments, else this pointer is not used and can be NULL.

The result of the key derivation is set in the `key_descriptor_derived` structure and consist in a new key id and the public data is exported if the public data and size are set in the `buffer` field.

3.1.4.7 struct smw_kdf_tls12_args

struct **smw_kdf_tls12_args**
Key derivation function TLS 1.2 arguments

3.1.4.7.1 Definition

```
struct smw_kdf_tls12_args {
    smw_tls12_kea_t key_exchange_name;
    smw_tls12_enc_t encryption_name;
    smw_hash_algo_t prf_name;
    bool ext_master_key;
    unsigned char *kdf_input;
    unsigned int kdf_input_length;
```

(continues on next page)

(continued from previous page)

```
    unsigned int master_sec_key_id;
    unsigned int client_w_enc_key_id;
    unsigned int server_w_enc_key_id;
    unsigned int client_w_mac_key_id;
    unsigned int server_w_mac_key_id;
    unsigned char *client_w_iv;
    unsigned int client_w_iv_length;
    unsigned char *server_w_iv;
    unsigned int server_w_iv_length;
}
```

3.1.4.7.2 Members

key_exchange_name
: Name of the key exchange algorithm. See *typedef smw_tls12_kea_t*

encryption_name
: Name of the encryption algorithm. See *typedef smw_tls12_enc_t*

prf_name
: Name of the Pseudo-Random Function (PRF). See *typedef smw_hash_algo_t*

ext_master_key
: If true, generates an extended master secret key

kdf_input
: Key derivation input data used to generate the master secret key

kdf_input_length
: Length in bytes of the `kdf_input` buffer

master_sec_key_id
: Generated master key identifier

client_w_enc_key_id
: Generated client write encryption key identifier

server_w_enc_key_id
: Generated server write encryption key identifier

client_w_mac_key_id
: Generated client write MAC key identifier (see note 1)

server_w_mac_key_id
: Generated server write MAC key identifier (see note 1)

client_w_iv
: Pointer to the Client IV buffer (see note 2)

client_w_iv_length
: Length of `client_w_iv` in bytes (see note 2)

server_w_iv
: Pointer to the Server IV buffer (see note 2)

server_w_iv_length
Length of `server_w_iv` in bytes (see note 2)

3.1.4.7.3 Description

This structure defines the additional arguments needed for the TLS 1.2 Key derivation (&smw_derive_key_args->kdf_name = *TLS12_KEY_EXCHANGE*).

Note 1: Client/Server write MAC key are not generated with AES GCM cipher encryption.

Note 2: Client/Server write IVs are generated only in case of Authentication Encryption with Additional Data Cipher mode (like AES CCM or GCM).

The key derivation *smw_derive_key_args->key_descriptor_derived* is filled only if the `key_exchange_name` request for an ephemeral public key. Following *smw_derive_key_args->key_descriptor_derived* fields are filled:

- `id`: set to 0
- `type_name`: Set the key type name
- `security_size`: Size in bits of the derived key
- `buffer`: Public key data buffer only

3.1.4.8 struct smw_update_key_args

struct **smw_update_key_args**
Key update arguments

3.1.4.8.1 Definition

```
struct smw_update_key_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
}
```

3.1.4.8.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

3.1.4.9 struct smw_import_key_args

struct **smw_import_key_args**

Key import arguments

3.1.4.9.1 Definition

```
struct smw_import_key_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    unsigned char *key_attributes_list;
    unsigned int key_attributes_list_length;
    struct smw_key_descriptor *key_descriptor;
}
```

3.1.4.9.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_attributes_list
Key attributes list. See *typedef smw_attr_key_type_t*

key_attributes_list_length
Length of a Key attributes list

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

3.1.4.9.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default Secure Subsystem configured for this Security Operation is used. The `key_descriptor` fields `type_name` and `security_size` must be given as input to define the type of key to import. The `key_descriptor` field `buffer` is mandatory. A public key, a private key or a key pair is imported if the corresponding pointer and size is set. The `key_descriptor` field `id`, if set by the caller (other than 0) will be the created key identifier on operation success. Else the API will returned a new key identifier if `id` is set as 0. The `buffer` field `format_name` is optional. The default value is "HEX".

3.1.4.10 struct smw_export_key_args

struct **smw_export_key_args**

Key export arguments

3.1.4.10.1 Definition

```
struct smw_export_key_args {
    unsigned char version;
    struct smw_key_descriptor *key_descriptor;
}
```

3.1.4.10.2 Members

version
Version of this structure

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

3.1.4.10.3 Description

The `key_descriptor` fields `id` must be given as input. The `key_descriptor` field `buffer` is mandatory. The public key buffer must be set in order to export the public Key, in case of asymmetric Keys. The private key buffer must be set in order to export the private Key, only if the Secure Subsystem supports it. In that case, the private Key may be encrypted, not plaintext. The user can use *smw_get_key_buffers_lengths()* to set correct lengths for the public/private key buffer(s).

3.1.4.11 struct smw_delete_key_args

struct **smw_delete_key_args**

Key deletion arguments

3.1.4.11.1 Definition

```
struct smw_delete_key_args {
    unsigned char version;
    struct smw_key_descriptor *key_descriptor;
    unsigned char *key_attributes_list;
    unsigned int key_attributes_list_length;
}
```

3.1.4.11.2 Members

version
    Version of this structure (must be equal 1).

key_descriptor
    Pointer to a Key descriptor object. See *struct smw_key_descriptor*

key_attributes_list
    Key attributes list. See *typedef smw_attr_key_type_t*

key_attributes_list_length
    Length of a Key attributes list

3.1.4.11.3 Description

The arguments `key_attributes_list` and `key_attributes_list_length` are supported since structure `version`=1.

Only the "FLUSH_KEY" key attribute is handled in the `key_attributes_list`.

The `key_descriptor` fields `id` must be given as input. The `key_descriptor` fields `buffer` is ignored.

3.1.4.12 struct smw_get_key_attributes_args

struct **smw_get_key_attributes_args**
    Get key attributes arguments

3.1.4.12.1 Definition

```
struct smw_get_key_attributes_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_key_descriptor *key_descriptor;
    smw_keymgr_privacy_t key_privacy;
    smw_keymgr_persistence_t persistence;
    unsigned char *policy_list;
    unsigned int policy_list_length;
    unsigned char *lifecycle_list;
    unsigned int lifecycle_list_length;
    unsigned int storage;
}
```

3.1.4.12.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

key_descriptor
Pointer to a Key descriptor object. See *struct smw_key_descriptor*

key_privacy
Key privacy type.

persistence
Key persistence.

policy_list
Key policy list. More details in *Key policy* of *typedef smw_attr_key_type_t*

policy_list_length
Length of the `policy_list` string.

lifecycle_list
Key lifecycle list.

lifecycle_list_length
Length of the `lifecycle_list`.

storage
Key storage identifier

3.1.4.12.3 Description

The `key_descriptor` fields `id` must be given as input. The `key_descriptor` fields `buffer` is ignored. The `key_descriptor` fields `type_name` and `security_size` are output.

Both `policy_list` and `lifecycle_list` are allocated by the operation *smw_get_key_attributes()* if retrieved and must be freed by user.

3.1.4.13 struct smw_commit_key_storage_args

struct **smw_commit_key_storage_args**
Commit non-volatile key storage arguments

3.1.4.13.1 Definition

```
struct smw_commit_key_storage_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
}
```

3.1.4.13.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

3.1.4.14 smw_generate_key

enum *smw_status_code* **smw_generate_key**(struct *smw_generate_key_args* *args)

Generate a Key.

Parameters

- **args** (struct *smw_generate_key_args**) – Pointer to the structure that contains the Key generation arguments.

3.1.4.14.1 Description

This function generates a Key.

3.1.4.14.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.15 smw_derive_key

enum *smw_status_code* **smw_derive_key**(struct *smw_derive_key_args* *args)

Derive a Key.

Parameters

- **args** (struct *smw_derive_key_args**) – Pointer to the structure that contains the Key derivation arguments.

3.1.4.15.1 Description

This function derives a Key.

3.1.4.15.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.16 smw_update_key

enum *smw_status_code* **smw_update_key**(struct *smw_update_key_args* *args)

Update a Key.

Parameters

- **args** (struct *smw_update_key_args**) – Pointer to the structure that contains the Key update arguments.

3.1.4.16.1 Description

This function updates the Key attribute list.

3.1.4.16.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.17 smw_import_key

enum *smw_status_code* **smw_import_key**(struct *smw_import_key_args* *args)

Import a Key.

Parameters

- **args** (struct *smw_import_key_args**) – Pointer to the structure that contains the Key import arguments.

3.1.4.17.1 Description

This function imports a Key into the storage managed by the Secure Subsystem. The key must be plain text.

3.1.4.17.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.18 smw_export_key

enum *smw_status_code* **smw_export_key**(struct *smw_export_key_args* *args)

Export a Key.

Parameters

- **args** (struct *smw_export_key_args**) – Pointer to the structure that contains the Key export arguments.

3.1.4.18.1 Description

This function exports a Key.

3.1.4.18.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.19 smw_delete_key

enum *smw_status_code* **smw_delete_key**(struct *smw_delete_key_args* *args)

Delete a Key.

Parameters

- **args** (struct *smw_delete_key_args**) – Pointer to the structure that contains the Key deletion arguments.

3.1.4.19.1 Description

This function deletes a Key.

3.1.4.19.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.20 smw_get_key_buffers_lengths

enum *smw_status_code* **smw_get_key_buffers_lengths**(struct *smw_key_descriptor* *descriptor)

Gets Key buffers lengths.

Parameters

- **descriptor** (struct *smw_key_descriptor**) – Pointer to the Key descriptor.

3.1.4.20.1 This function calculates either

- The subsystem key buffers' lengths of the given `descriptor` field `id`. Only the exportable buffer lengths are returned.
- The standard key buffers's lengths of the given `descriptor` fields `type_name`, `security_size`.

The `descriptor` field `buffer` is mandatory. The `buffer` field `format_name` is optional. The `buffer` fields `public_length`, `modulus` and `private_length` are updated.

3.1.4.20.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.21 smw_get_key_type_name

enum *smw_status_code* **smw_get_key_type_name**(struct *smw_key_descriptor* *descriptor)

Gets the Key type name.

Parameters

- **descriptor** (struct *smw_key_descriptor**) – Pointer to the Key descriptor.

3.1.4.21.1 Description

This function gets the Key type name given the Key ID. The `descriptor` field `id` must be given as input. The `descriptor` fields `type_name` is updated.

3.1.4.21.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.4.22 smw_get_security_size

enum *smw_status_code* **`smw_get_security_size`**(struct *smw_key_descriptor* *descriptor)

Gets the Security size.

Parameters

- **`descriptor`** (struct *smw_key_descriptor**) – Pointer to the Key descriptor.

3.1.4.22.1 Description

This function gets the Security size given the Key ID. The `descriptor` field `id` must be given as input. The `descriptor` fields `security_size` is updated.

3.1.4.22.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.4.23 smw_get_key_attributes

enum *smw_status_code* **`smw_get_key_attributes`**(struct *smw_get_key_attributes_args* *args)

Get the key attributes.

Parameters

- **`args`** (struct *smw_get_key_attributes_args**) – Pointer to the structure that contains the Key attributes arguments.

3.1.4.23.1 Description

This function gets the Key attributes retrieved for the subsystem owning the given key identifier. If some key attributes are not supported, the output values are empty.

3.1.4.23.2 Return

See *enum smw_status_code*

- Common return codes

3.1.4.24 smw_commit_key_storage

enum *smw_status_code* **smw_commit_key_storage**(struct *smw_commit_key_storage_args* *args)

Commit the active non-volatile key storage

Parameters

- **args** (struct *smw_commit_key_storage_args**) – Pointer to the structure that contains the commit storage arguments.

3.1.4.24.1 Description

This function ensures that the non-volatile key storage opened by the subsystem is pushed in physical memory and associated anti-rollback protection counter is incremented.

3.1.4.24.2 Return

See *enum smw_status_code*

- Common return codes

3.1.5 Device management APIs

3.1.5.1 Introduction

The device APIs allow user of the library to:

- Get information about the device.
- Change the device lifecycle.

3.1.5.2 struct smw_device_attestation_args

struct **smw_device_attestation_args**

Device attestation arguments

3.1.5.2.1 Definition

```
struct smw_device_attestation_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    unsigned char *challenge;
    unsigned int challenge_length;
    unsigned char *certificate;
    unsigned int certificate_length;
}
```

3.1.5.2.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

challenge
Caller unique ephemeral value (e.g. nonce)

challenge_length
Length (in bytes) of the `challenge` value

certificate
Device attestation certificate.

certificate_length
Length (in bytes) of the `certificate`.

3.1.5.2.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

`challenge` length depends of the device (refer to the subsystem capabilities). If the length is bigger than expected, it will be cut to keep only the device maximum size. If the length is shorter, the challenge value will be completed with 0's.

3.1.5.3 struct smw_device_uuid_args

struct **smw_device_uuid_args**

Device UUID arguments

3.1.5.3.1 Definition

```
struct smw_device_uuid_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    unsigned char *certificate;
    unsigned int certificate_length;
    unsigned char *uuid;
    unsigned int uuid_length;
}
```

3.1.5.3.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

certificate
Device attestation certificate.

certificate_length
Length (in bytes) of the `certificate`.

uuid
Device UUID buffer

uuid_length
Length (in bytes) of the `uuid`

3.1.5.3.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

Two methods are allowed to get the Device UUID.

- **Method #1**
  Extract the device UUID from the device certificate. The Device Certificate (@certificate) is previously read using the *smw_device_attestation()* API.
- **Method #2**
  Read the device UUID without providing the Device Certificate. The field `certificate` must be set to NULL.

3.1.5.4 struct smw_device_lifecycle_args

struct **smw_device_lifecycle_args**

Device lifecycle arguments

3.1.5.4.1 Definition

```
struct smw_device_lifecycle_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    smw_lifecycle_t lifecycle_name;
}
```

3.1.5.4.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

lifecycle_name
Device lifecycle name. See *typedef smw_lifecycle_t*

3.1.5.4.3 Description

subsystem_name designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.5.5 smw_device_attestation

enum *smw_status_code* **smw_device_attestation**(struct *smw_device_attestation_args* *args)

Get the device attestation certificate.

Parameters

- **args** (struct *smw_device_attestation_args**) – Pointer to the structure that contains the device attestation arguments.

3.1.5.5.1 Description

Reads the device attestation certificate.

Certificate length `args` field is updated to the correct value when:

- Length is bigger than expected. In this case operation succeeded.
- Length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.
- Certificate buffer is set the NULL. In this case operation returned SMW_STATUS_OK

3.1.5.5.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.5.6 smw_device_get_uuid

enum *smw_status_code* **`smw_device_get_uuid`**(struct *smw_device_uuid_args* *args)

Get the device UUID.

Parameters

- **`args`** (`struct` *`smw_device_uuid_args`*`*`) – Pointer to the structure that contains the device UUID arguments.

3.1.5.6.1 Description

Extracts device UUID from the device certificate or reads the device UUID without device certificate.

Device UUID buffer is in big endian format.

UUID length `args` field is updated to the correct value when:

- Length is bigger than expected. In this case operation succeeded.
- Length is shorter than expected. In this case operation failed and returned SMW_STATUS_OUTPUT_TOO_SHORT.
- UUID buffer is set the NULL. In this case operation returned SMW_STATUS_OK

3.1.5.6.2 Return

See *`enum smw_status_code`*

- Common return codes

3.1.5.7 smw_device_set_lifecycle

enum *smw_status_code* **smw_device_set_lifecycle**(struct *smw_device_lifecycle_args* *args)

Set the device to given lifecycle.

Parameters

- **args** (struct *smw_device_lifecycle_args**) – Pointer to the structure that contains the device lifecycle arguments.

3.1.5.7.1 Description

Forward the device lifecycle to the given value. The device must be reset to propagate the new lifecycle.

Caution: Forwarding device lifecycle is not reversible.

3.1.5.7.2 Return

See *enum smw_status_code*

- Common return codes

3.1.5.8 smw_device_get_lifecycle

enum *smw_status_code* **smw_device_get_lifecycle**(struct *smw_device_lifecycle_args* *args)

Get the device active lifecycle.

Parameters

- **args** (struct *smw_device_lifecycle_args**) – Pointer to the structure that contains the device lifecycle arguments.

3.1.5.8.1 Return

See *enum smw_status_code*

- Common return codes

3.1.6 Storage APIs

3.1.6.1 Introduction

The storage APIs allow user of the library to:

- Store data.
- Retrieve data.
- Delete data.

The data storing operation allows user to request the subsystem to either:

- store data as given by the user, reading back the data will in the same format as given by user.

- encrypt data then store, reading back the data will be a blob of encrypted data.
- encrypt and sign data then store, reading back the data will be a signed blob of encrypted data.
- sign data then store, reading back the data will be a signed blob of data. Knowing that data format is identical as the given by user.

Refer to the subsystem capabilities for more details of the supported features and blob format.

Signature is limited to MAC signature.

3.1.6.2 struct smw_data_descriptor

struct **smw_data_descriptor**

Data descriptor

3.1.6.2.1 Definition

```
struct smw_data_descriptor {
    unsigned int identifier;
    unsigned char *data;
    unsigned int length;
    unsigned char *attributes_list;
    unsigned int attributes_list_length;
}
```

3.1.6.2.2 Members

identifier
Data identifier

data
Pointer to the data buffer

length
Length of buffer `data`

attributes_list
Data attributes list. See *typedef smw_attr_data_type_t*

attributes_list_length
Length of buffer `attributes_list`

3.1.6.3 struct smw_encryption_args

struct **smw_encryption_args**

Encryption arguments

3.1.6.3.1 Definition

```
struct smw_encryption_args {
    struct smw_key_descriptor **keys_desc;
    unsigned int nb_keys;
    smw_cipher_mode_t mode_name;
    unsigned char *iv;
    unsigned int iv_length;
}
```

3.1.6.3.2 Members

keys_desc
Pointer to an array of pointers to key descriptors. See *struct smw_key_descriptor*

nb_keys
Number of entries of `keys_desc`

mode_name
Cipher mode name. See *typedef smw_cipher_mode_t*

iv
Pointer to initialization vector

iv_length
`iv` length in bytes

3.1.6.3.3 Description

Depending on `mode_name`, the `iv` is optional and represents:

- Initialization Vector (CBC, CTS)
- Initial Counter Value (CTR)
- Tweak Value (XTS)

3.1.6.4 struct smw_sign_args

struct **smw_sign_args**

Sign arguments

3.1.6.4.1 Definition

```
struct smw_sign_args {
    struct smw_key_descriptor *key_descriptor;
    smw_mac_algo_t algo_name;
    smw_hash_algo_t hash_name;
}
```

3.1.6.4.2 Members

key_descriptor
Pointer to a signing Key descriptor object. See *struct smw_key_descriptor*

algo_name
MAC algorithm name. See *typedef smw_mac_algo_t*

hash_name
Hash algorithm name. See *typedef smw_hash_algo_t*

3.1.6.5 struct smw_store_data_args

struct **smw_store_data_args**

Store data arguments

3.1.6.5.1 Definition

```
struct smw_store_data_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_data_descriptor *data_descriptor;
    struct smw_encryption_args *encryption_args;
    struct smw_sign_args *sign_args;
}
```

3.1.6.5.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

data_descriptor
Data descriptor. See *struct smw_data_descriptor*

encryption_args
Encryption arguments. See *struct smw_encryption_args*

sign_args
Sign arguments. See *struct smw_sign_args*

3.1.6.5.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

The `encryption_args` and `smw_sign_args` arguments are optional. If defined the operation consists respectively in encrypting and/or signing the data. The capability to encrypt and/or sign data is function of the subsystem. Refer to the *Subsystems Capabilities*.

3.1.6.6 struct smw_retrieve_data_args

struct **smw_retrieve_data_args**

Retrieve data arguments

3.1.6.6.1 Definition

```
struct smw_retrieve_data_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_data_descriptor *data_descriptor;
}
```

3.1.6.6.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

data_descriptor
Data descriptor. See *struct smw_data_descriptor*

3.1.6.6.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.6.7 struct smw_delete_data_args

struct **smw_delete_data_args**

Delete data arguments

3.1.6.7.1 Definition

```
struct smw_delete_data_args {
    unsigned char version;
    smw_subsystem_t subsystem_name;
    struct smw_data_descriptor *data_descriptor;
}
```

3.1.6.7.2 Members

version
Version of this structure

subsystem_name
Secure Subsystem name. See *typedef smw_subsystem_t*

data_descriptor
Data descriptor. See *struct smw_data_descriptor*

3.1.6.7.3 Description

`subsystem_name` designates the Secure Subsystem to be used. If this field is NULL, the default configured Secure Subsystem is used.

3.1.6.8 smw_store_data

enum *smw_status_code* **smw_store_data**(struct *smw_store_data_args* *args)

Store data.

Parameters

- **args** (struct *smw_store_data_args**) – Pointer to the structure that contains the store data arguments.

3.1.6.8.1 Description

Stores the data.

3.1.6.8.2 Return

See *enum smw_status_code*

- Common return codes

3.1.6.9 smw_retrieve_data

enum *smw_status_code* **smw_retrieve_data**(struct *smw_retrieve_data_args* *args)

Retrieve data.

Parameters

- **args** (struct *smw_retrieve_data_args**) – Pointer to the structure that contains the retrieve data arguments.

3.1.6.9.1 Description

Retrieves the data.

3.1.6.9.2 Return

See *enum smw_status_code*

- Common return codes
- SMW_STATUS_DATA_ALREADY_RETRIEVED

3.1.6.10 smw_delete_data

enum *smw_status_code* **smw_delete_data**(struct *smw_delete_data_args* *args)

Delete data.

Parameters

- **args** (struct *smw_delete_data_args**) – Pointer to the structure that contains the store data arguments.

3.1.6.10.1 Description

Deletes the data.

3.1.6.10.2 Return

See *enum smw_status_code*

- Common return codes

3.1.7 Return codes

3.1.7.1 enum smw_status_code

enum **smw_status_code**

Security Middleware status codes

3.1.7.1.1 Definition

```
enum smw_status_code {
    SMW_STATUS_OK,
    SMW_STATUS_INVALID_VERSION,
    SMW_STATUS_INVALID_BUFFER,
    SMW_STATUS_EOF,
    SMW_STATUS_SYNTAX_ERROR,
    SMW_STATUS_UNKNOWN_NAME,
    SMW_STATUS_UNKNOWN_ID,
    SMW_STATUS_TOO_LARGE_NUMBER,
    SMW_STATUS_ALLOC_FAILURE,
    SMW_STATUS_INVALID_PARAM,
    SMW_STATUS_VERSION_NOT_SUPPORTED,
    SMW_STATUS_SUBSYSTEM_LOAD_FAILURE,
    SMW_STATUS_SUBSYSTEM_UNLOAD_FAILURE,
    SMW_STATUS_SUBSYSTEM_FAILURE,
    SMW_STATUS_SUBSYSTEM_NOT_CONFIGURED,
    SMW_STATUS_OPERATION_NOT_SUPPORTED,
    SMW_STATUS_OPERATION_NOT_CONFIGURED,
    SMW_STATUS_OPERATION_FAILURE,
    SMW_STATUS_SIGNATURE_INVALID,
    SMW_STATUS_NO_KEY_BUFFER,
    SMW_STATUS_OUTPUT_TOO_SHORT,
    SMW_STATUS_SIGNATURE_LEN_INVALID,
    SMW_STATUS_OPS_INVALID,
    SMW_STATUS_MUTEX_INIT_FAILURE,
    SMW_STATUS_MUTEX_DESTROY_FAILURE,
    SMW_STATUS_INVALID_TAG,
    SMW_STATUS_RANGE_DUPLICATE,
    SMW_STATUS_ALGO_NOT_CONFIGURED,
```

(continues on next page)

```
    SMW_STATUS_CONFIG_ALREADY_LOADED,
    SMW_STATUS_NO_CONFIG_LOADED,
    SMW_STATUS_SUBSYSTEM_OUT_OF_MEMORY,
    SMW_STATUS_SUBSYSTEM_STORAGE_NO_SPACE,
    SMW_STATUS_SUBSYSTEM_STORAGE_ERROR,
    SMW_STATUS_SUBSYSTEM_CORRUPT_OBJECT,
    SMW_STATUS_LOAD_METHOD_DUPLICATE,
    SMW_STATUS_LIBRARY_ALREADY_INIT,
    SMW_STATUS_SUBSYSTEM_LOADED,
    SMW_STATUS_SUBSYSTEM_NOT_LOADED,
    SMW_STATUS_OBJ_DB_INIT,
    SMW_STATUS_OBJ_DB_CREATE,
    SMW_STATUS_OBJ_DB_UPDATE,
    SMW_STATUS_OBJ_DB_DELETE,
    SMW_STATUS_OBJ_DB_GET_INFO,
    SMW_STATUS_KEY_DB_INIT,
    SMW_STATUS_KEY_DB_CREATE,
    SMW_STATUS_KEY_DB_UPDATE,
    SMW_STATUS_KEY_DB_DELETE,
    SMW_STATUS_KEY_DB_GET_INFO,
    SMW_STATUS_KEY_POLICY_ERROR,
    SMW_STATUS_KEY_POLICY_WARNING_IGNORED,
    SMW_STATUS_KEY_INVALID,
    SMW_STATUS_MUTEX_LOCK_FAILURE,
    SMW_STATUS_MUTEX_UNLOCK_FAILURE,
    SMW_STATUS_INVALID_LIBRARY_CONTEXT,
    SMW_STATUS_INVALID_CONFIG_DATABASE,
    SMW_STATUS_DATA_ALREADY_RETRIEVED,
    SMW_STATUS_INVALID_LIFECYCLE
};
```

3.1.7.1.2 Constants

SMW_STATUS_OK
: Function returned successfully.

SMW_STATUS_INVALID_VERSION
: The version of the configuration file is not supported.

SMW_STATUS_INVALID_BUFFER
: The configuration file passed by OSAL to the library is not valid.

SMW_STATUS_EOF
: The configuration file is syntactically too short.

SMW_STATUS_SYNTAX_ERROR
: The configuration file is syntactically wrong.

SMW_STATUS_UNKNOWN_NAME
: One of the string name arguments is not valid.

SMW_STATUS_UNKNOWN_ID
One of the identifier arguments is not valid.

SMW_STATUS_TOO_LARGE_NUMBER
The configuration file defines a too big numeral value.

SMW_STATUS_ALLOC_FAILURE
Internal allocation failure.

SMW_STATUS_INVALID_PARAM
One of the argument parameter is not valid.

SMW_STATUS_VERSION_NOT_SUPPORTED
Argument version not compatible.

SMW_STATUS_SUBSYSTEM_LOAD_FAILURE
Load of the Secure Subsystem failed.

SMW_STATUS_SUBSYSTEM_UNLOAD_FAILURE
Unload of the Secure Subsystem failed.

SMW_STATUS_SUBSYSTEM_FAILURE
Secure Subsystem operation general failure.

SMW_STATUS_SUBSYSTEM_NOT_CONFIGURED
Secure Subsystem is not configured in the user configuration.

SMW_STATUS_OPERATION_NOT_SUPPORTED
Operation is not supported by the Secure Subsystem.

SMW_STATUS_OPERATION_NOT_CONFIGURED
Operation is not configured in the user configuration.

SMW_STATUS_OPERATION_FAILURE
Operation general failure. Error returned before calling the Secure Subsystem.

SMW_STATUS_SIGNATURE_INVALID
The Signature is not valid.

SMW_STATUS_NO_KEY_BUFFER
No Key buffer is set in the Key descriptor structure.

SMW_STATUS_OUTPUT_TOO_SHORT
Output buffer is too small. Output size field is updated with the expected size.

SMW_STATUS_SIGNATURE_LEN_INVALID
The Signature length is not valid.

SMW_STATUS_OPS_INVALID
OSAL operations structure is invalid.

SMW_STATUS_MUTEX_INIT_FAILURE
Mutex initalization has failed.

SMW_STATUS_MUTEX_DESTROY_FAILURE
Mutex destruction has failed.

SMW_STATUS_INVALID_TAG
Tag is invalid.

SMW_STATUS_RANGE_DUPLICATE
Size range is defined more than once for a given algorithm.

SMW_STATUS_ALGO_NOT_CONFIGURED
Size range is defined but the corresponding algorithm is not configured.

SMW_STATUS_CONFIG_ALREADY_LOADED
User configuration is already loaded. To load another one, the Unload configuration API must be called first.

SMW_STATUS_NO_CONFIG_LOADED
No user configuration is loaded.

SMW_STATUS_SUBSYSTEM_OUT_OF_MEMORY
Subsystem memory allocation failure.

SMW_STATUS_SUBSYSTEM_STORAGE_NO_SPACE
Not enough space in the secure subsystem to handle the requested operation.

SMW_STATUS_SUBSYSTEM_STORAGE_ERROR
Generic secure subsystem storage error.

SMW_STATUS_SUBSYSTEM_CORRUPT_OBJECT
An object stored in the secure subsystem is corrupted.

SMW_STATUS_LOAD_METHOD_DUPLICATE
The load/unload method is defined more than once.

SMW_STATUS_LIBRARY_ALREADY_INIT
Library is already initialized.

SMW_STATUS_SUBSYSTEM_LOADED
Secure Subsystem is loaded.

SMW_STATUS_SUBSYSTEM_NOT_LOADED
Secure Subsystem is not loaded.

SMW_STATUS_OBJ_DB_INIT
Initialization error of the object database.

SMW_STATUS_OBJ_DB_CREATE
Object database creation error.

SMW_STATUS_OBJ_DB_UPDATE
Object database update error.

SMW_STATUS_OBJ_DB_DELETE
Object database delete error.

SMW_STATUS_OBJ_DB_GET_INFO
Object database get information error.

SMW_STATUS_KEY_DB_INIT
Initialization error of the key database.

SMW_STATUS_KEY_DB_CREATE
Key database creation error.

SMW_STATUS_KEY_DB_UPDATE
Key database update error.

SMW_STATUS_KEY_DB_DELETE
Key database delete error.

SMW_STATUS_KEY_DB_GET_INFO
Key database get information error.

SMW_STATUS_KEY_POLICY_ERROR
The key policy is syntactically wrong.

SMW_STATUS_KEY_POLICY_WARNING_IGNORED
At least one element of the key policy is ignored.

SMW_STATUS_KEY_INVALID
Key used for the operation is not valid.

SMW_STATUS_MUTEX_LOCK_FAILURE
Mutex lock has failed.

SMW_STATUS_MUTEX_UNLOCK_FAILURE
Mutex unlock has failed.

SMW_STATUS_INVALID_LIBRARY_CONTEXT
Library context is not valid.

SMW_STATUS_INVALID_CONFIG_DATABASE
Configuration database is not valid.

SMW_STATUS_DATA_ALREADY_RETRIEVED
The data was read once and has been already retrieved.

SMW_STATUS_INVALID_LIFECYCLE
Device lifecycle not valid, or object not accessible in current device lifecyle.

3.1.7.1.3 Status code classification

- Common return codes
  - SMW_STATUS_OK
  - SMW_STATUS_UNKNOWN_NAME
  - SMW_STATUS_UNKNOWN_ID
  - SMW_STATUS_ALLOC_FAILURE
  - SMW_STATUS_INVALID_PARAM
  - SMW_STATUS_VERSION_NOT_SUPPORTED
  - SMW_STATUS_SUBSYSTEM_LOAD_FAILURE
  - SMW_STATUS_SUBSYSTEM_UNLOAD_FAILURE
  - SMW_STATUS_SUBSYSTEM_FAILURE
  - SMW_STATUS_SUBSYSTEM_NOT_CONFIGURED
  - SMW_STATUS_OPERATION_NOT_SUPPORTED
  - SMW_STATUS_OPERATION_NOT_CONFIGURED
  - SMW_STATUS_OPERATION_FAILURE

- SMW_STATUS_NO_KEY_BUFFER
- SMW_STATUS_OUTPUT_TOO_SHORT
- SMW_STATUS_SUBSYSTEM_OUT_OF_MEMORY
- SMW_STATUS_SUBSYSTEM_STORAGE_NO_SPACE
- SMW_STATUS_SUBSYSTEM_STORAGE_ERROR
- SMW_STATUS_SUBSYSTEM_CORRUPT_OBJECT
- SMW_STATUS_SUBSYSTEM_LOADED
- SMW_STATUS_SUBSYSTEM_NOT_LOADED
- SMW_STATUS_KEY_INVALID
- SMW_STATUS_INVALID_LIFECYCLE

- Specific return codes - Library initialization
  - SMW_STATUS_OPS_INVALID
  - SMW_STATUS_MUTEX_INIT_FAILURE
  - SMW_STATUS_MUTEX_DESTROY_FAILURE
  - SMW_STATUS_LIBRARY_ALREADY_INIT
  - SMW_STATUS_MUTEX_LOCK_FAILURE
  - SMW_STATUS_MUTEX_UNLOCK_FAILURE
  - SMW_STATUS_INVALID_LIBRARY_CONTEXT
  - SMW_STATUS_INVALID_CONFIG_DATABASE
- Specific return codes - Configuration file
  - SMW_STATUS_INVALID_VERSION
  - SMW_STATUS_INVALID_BUFFER
  - SMW_STATUS_EOF
  - SMW_STATUS_SYNTAX_ERROR
  - SMW_STATUS_TOO_LARGE_NUMBER
  - SMW_STATUS_INVALID_TAG
  - SMW_STATUS_RANGE_DUPLICATE
  - SMW_STATUS_ALGO_NOT_CONFIGURED
  - SMW_STATUS_CONFIG_ALREADY_LOADED
  - SMW_STATUS_NO_CONFIG_LOADED
  - SMW_STATUS_LOAD_METHOD_DUPLICATE
- Specific return codes - Signature
  - SMW_STATUS_SIGNATURE_INVALID
  - SMW_STATUS_SIGNATURE_LEN_INVALID

- **Specific return codes - Object database**
  - SMW_STATUS_ERROR_OBJ_DB_INIT
  - SMW_STATUS_ERROR_OBJ_DB_CREATE
  - SMW_STATUS_ERROR_OBJ_DB_UPDATE
  - SMW_STATUS_ERROR_OBJ_DB_DELETE
  - SMW_STATUS_ERROR_OBJ_DB_GET_INFO
- **Specific return codes - Key database**
  - SMW_STATUS_ERROR_KEY_DB_INIT
  - SMW_STATUS_ERROR_KEY_DB_CREATE
  - SMW_STATUS_ERROR_KEY_DB_UPDATE
  - SMW_STATUS_ERROR_KEY_DB_DELETE
  - SMW_STATUS_ERROR_KEY_DB_GET_INFO
- **Specific return codes - Key manager**
  - SMW_STATUS_KEY_POLICY_ERROR
  - SMW_STATUS_KEY_POLICY_WARNING_IGNORED
- **Specific return codes - Data storage**
  - SMW_STATUS_DATA_ALREADY_RETRIEVED

3.1.8 Strings APIs

3.1.8.1 typedef smw_subsystem_t

type **`smw_subsystem_t`**

Subsystem name

3.1.8.1.1 Values

- TEE
- HSM

3.1.8.2 typedef smw_key_type_t

type **`smw_key_type_t`**

Key type name

3.1.8.2.1 Values

- NIST
- BRAINPOOL_R1
- BRAINPOOL_T1
- AES
- DES
- DES3
- DSA_SM2_FP
- SM4
- HMAC
- HMAC_MD5
- HMAC_SHA1
- HMAC_SHA224
- HMAC_SHA256
- HMAC_SHA384
- HMAC_SHA512
- HMAC_SM3
- RSA

3.1.8.3 typedef smw_keymgr_privacy_t

type `smw_keymgr_privacy_t`

Key privacy name

3.1.8.3.1 Values

- PUBLIC
- PRIVATE
- KEYPAIR

3.1.8.4 typedef smw_keymgr_persistence_t

type **smw_keymgr_persistence_t**

Key persistence name

3.1.8.4.1 Values

- TRANSIENT
- PERSISTENT
- PERMANENT

3.1.8.5 typedef smw_hash_algo_t

type **smw_hash_algo_t**

Hash algorithm name

3.1.8.5.1 Values

- MD5
- SHA1
- SHA224
- SHA256
- SHA384
- SHA512
- SM3

3.1.8.6 typedef smw_mac_algo_t

type **smw_mac_algo_t**

MAC algorithm name

3.1.8.6.1 Values

- CMAC
- CMAC_TRUNCATED
- HMAC
- HMAC_TRUNCATED

3.1.8.7 typedef smw_cipher_mode_t

type **smw_cipher_mode_t**

Cipher mode name

3.1.8.7.1 Values

- CBC
- CTR
- CTS
- ECB
- XTS

3.1.8.8 typedef smw_aead_mode_t

type **smw_aead_mode_t**

AEAD mode name

3.1.8.8.1 Values

- CCM
- GCM

3.1.8.9 typedef smw_cipher_operation_t

type **smw_cipher_operation_t**

Cipher operation name

3.1.8.9.1 Values

- ENCRYPT
- DECRYPT

3.1.8.10 typedef smw_aead_operation_t

type **smw_aead_operation_t**

AEAD operation name

3.1.8.10.1 Values

- ENCRYPT
- DECRYPT

3.1.8.11 typedef smw_key_format_t

type **smw_key_format_t**

Key format name

3.1.8.11.1 Values

- HEX: hexadecimal value (no encoding)
- BASE64: base 64 encoding value

3.1.8.12 typedef smw_attr_key_type_t

type **smw_attr_key_type_t**

Key definition attribute type name

3.1.8.12.1 Description

An attribute is encoded with a Type-Length-Value (TLV) format. Function of the attribute type, the TLV scheme varies. Refer to *TLV coding*

3.1.8.12.2 Key Manager attributes

The following Table 3.1 lists all TLV attributes supported by key manager operations like generate, import, derive, delete.

Table 3.1: Key manager attributes

Type Value	Encoding	Description
PERSISTENT	boolean	If present key is persistent.
RSA_PUB_EXP	numeral	Setup the RSA Public exponent value. The default value is 65537 if this attribute is not defined.
FLUSH_KEY	boolean	If present, ensure that the key storage is up to date.
POLICY	variable length list	This attribute is used to restrict the key usage(s) and algorithm(s). The following *Key policy* details how a key policy is defined.
STORAGE_ID	numeral	Subsystem storage identifier. EdgeLock 2GO storage identifiers: • Key object: NXP_EL2GO_KEY • Data object: NXP_EL2GO_DATA

3.1.8.12.2.1 Key policy

The key policy is built with a TLV variable length list in which one or more key usage(s) are listed. To each key usage, algorithm(s) might be restricted.

This attribute may or may not be significative (fully or partially) function of the subsystem handling the key. Refer to the *Subsystems Capabilities* for more details.

3.1.8.12.3 Signature attributes

The following Table 3.2 lists all TLV attributes supported by sign and verify operations.

Table 3.2: Signature attributes

Type Value	Encoding	Description
SIGNATURE_TYPE	string	Define the type of signature in case multiple options are possible. Otherwise the signature type is function of the key type. Refer to *smw_signature_type_t* to get the possible attribute value.
SALT_LENGTH	string	If signature is RSASSA-PSS, set the salt length of the signature.
TLS_MAC_FINISH	string	Define the TLS finish message signature type to generate. Value is either “CLIENT” or “SERVER” corresponding to client or server finish signature.

3.1.8.13 typedef smw_attr_data_type_t

type **`smw_attr_data_type_t`**

Data definition attribute type name

3.1.8.13.1 Description

An attribute is encoded with a Type-Length-Value (TLV) format. Function of the attribute type, the TLV scheme varies. Refer to *TLV coding*

The following Table 3.3 lists all TLV attributes supported by data manager store operation.

Table 3.3: Data manager attributes

Type Value	Encoding	Description
READ_ONLY	boolean	Data is read-only.
READ_ONCE	boolean	Data is read once time, when data is retrieved, data is deleted.
LIFECYCLE	variable length list	This attribute is used to restrict the data accessibility. The following *Data lifecycle* gives more details.

3.1.8.13.1.1 Data lifecycle

The data lifecycle is built with a TLV variable length list in which one or more string below. This attribute limits the access of the data in the corresponding device lifecycle.

The CURRENT string value means that data is accessible only in the current device lifecycle when data is created.

Table 3.4: Data lifecyle attribute

String Value
OPEN
CLOSED
CLOSED_LOCKED
CURRENT

This attribute may or may not be significative (fully or partially) function of the subsystem handling the data. Refer to the *Subsystems Capabilities* for more details.

3.1.8.14 typedef smw_signature_type_t

type **smw_signature_type_t**

Signature type name

3.1.8.14.1 Values

- DEFAULT
- RSASSA-PKCS1-V1_5
- RSASSA-PSS

3.1.8.15 typedef smw_kdf_t

type **smw_kdf_t**

Key derivation function name

3.1.8.15.1 Values

- TLS12_KEY_EXCHANGE

3.1.8.16 typedef smw_tls12_kea_t

type **smw_tls12_kea_t**

TLS 1.2 Key exchange algorithm name

3.1.8.16.1 Values

- DH_DSS
- DH_RSA
- DHE_DSS
- DHE_RSA
- ECDH_ECDSA
- ECDH_RSA
- ECDHE_ECDSA
- ECDHE_RSA
- RSA

3.1.8.17 typedef smw_tls12_enc_t

type **smw_tls12_enc_t**

TLS 1.2 encryption algorithm name

3.1.8.17.1 Values

- 3DES_EDE_CBC
- AES_128_CBC
- AES_128_GCM
- AES_256_CBC
- AES_256_GCM
- RC4_128

3.1.8.18 typedef smw_lifecycle_t

type **smw_lifecycle_t**

Device lifecycle name

3.1.8.18.1 Values

- OPEN
- CLOSED
- CLOSED_LOCKED
- OEM_RETURN
- NXP_RETURN

3.1.9 Examples

3.1.9.1 Authentication Encryption/Decryption (AEAD)

3.1.9.1.1 Example 1: AEAD one-shot encryption operation

```
#define IV_LEN 12
#define AAD_LEN 20
#define DATA_LEN 32
#define CIPHER_LEN 32
#define TAG_LEN 16

int main(int argc, char *argv[])
{
    int res = SMW_STATUS_OPERATION_FAILURE;
```

(continues on next page)

(continued from previous page)

```
    unsigned char iv[IV_LEN] = {...};
    unsigned char aad[AAD_LEN] = {...};
    unsigned char input[DATA_LEN] = {...};
    unsigned char output[CIPHER_LEN + TAG_LEN] = {0};

    struct smw_aead_args args = {0};
    struct smw_aead_data_args data_args = {0};
    struct smw_aead_init_args init_args = {0};
    struct smw_key_descriptor key_desc = {0};

    init_args.subsystem_name = "TEE";
    init_args.operation_name = "ENCRYPT";
    init_args.mode_name = "GCM";
    init_args.aad_length = AAD_LEN;
    init_args.tag_length = TAG_LEN;
    init_args.plaintext_length = DATA_LEN;
    init_args.key_desc = &key_desc;

    data_args.input_length = DATA_LEN;
    data_args.input = input;
    data_args.output_length = CIPHER_LEN + TAG_LEN;
    data_args.output = output;

    // One-shot AE encryption operation
    args.init = init_args;
    args.data = data_args;
    args.aad = aad;
    res = smw_aead(&args);
    return res;
}
```

3.1.9.1.2 Example 2: AEAD multi-part encryption operation

```
#define IV_LEN 12
#define AAD_LEN 20
#define DATA_LEN 32
#define TAG_LEN 16
#define CIPHER_LEN 32

int main(int argc, char *argv[])
{

    int res = SMW_STATUS_OPERATION_FAILURE;

    unsigned char iv[IV_LEN] = {...};
    unsigned char aad[AAD_LEN] = {...};
    unsigned char input[DATA_LEN] = {...};
    unsigned char output[CIPHER_LEN + TAG_LEN] = {0};
```

(continues on next page)

(continued from previous page)

```
struct smw_aead_args args = {0};
struct smw_aead_data_args data_args = {0};
struct smw_aead_aad_args aad_args = {0};
struct smw_aead_final_args final_args = {0};
struct smw_aead_init_args init_args = {0};
struct smw_key_descriptor key_desc = {0};
struct smw_op_context *op_ctx = 0;

init_args.subsystem_name = "TEE";
init_args.operation_name = "ENCRYPT";
init_args.mode_name = "GCM";
init_args.aad_length = AAD_LEN;
init_args.tag_length = TAG_LEN;
init_args.plaintext_length = DATA_LEN;

// Allocate memory to pointer to operation context
op_ctx = calloc(1, sizeof(*op_ctx));
init_args.context = op_ctx;
init_args.key_desc = &key_desc;

// Initialize multi-part AEAD operation
res = smw_aead_init(&init_args);
if (res != SMW_STATUS_OK)
    goto exit;

// Add additional data to an active AEAD operation.
aad_args.aad = aad;
aad_args.aad_length = AAD_LEN;
aad_args.context = init_args.context;
res = smw_aead_update_add(&aad_args);
if (res != SMW_STATUS_OK)
    goto exit;

/**
 * Encrypt 1st message fragment in an active
 * multi-part AEAD encryption operation.
 */
data_args.input_length = 16;
data_args.input = input;
data_args.output_length = 16;
data_args.output = output;
data_args.context = init_args.context;
res = smw_aead_update(&data_args);
if (res != SMW_STATUS_OK)
    goto exit;

/**
 * Encrypt 2nd message fragment in an active
```

(continues on next page)

(continued from previous page)

```
     * multi-part AEAD encryption operation.
     */
    data_args.input_length = 16;
    data_args.input = &input[16];
    data_args.output_length = 16;
    data_args.output = &output[16];
    data_args.context = init_args.context;
    res = smw_aead_update(&data_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    /**
     * Finish encrypting the message in an active
     * multi-part AEAD operation.
     */
    final_args.operation_name = "ENCRYPT";
    final_args.data.context = init_args.context;
    final_args.data.input = NULL;
    final_args.data.input_length = 0;
    final_args.data.output = &output[32];
    final_args.data.output_length = TAG_LEN;
    final_args.tag_length = TAG_LEN;
    res = smw_aead_final(&final_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    exit:
    if (op_ctx)
        free(op_ctx);

    return res;
}
```

3.1.9.1.3 Example 3: AEAD multi-part decryption operation

```
#define IV_LEN 12
#define AAD_LEN 20
#define DATA_LEN 32
#define TAG_LEN 16
#define CIPHER_LEN 32

int main(int argc, char *argv[])
{

    int res = SMW_STATUS_OPERATION_FAILURE;

    unsigned char iv[IV_LEN] = {...};
    unsigned char aad[AAD_LEN] = {...};
```

(continues on next page)

(continued from previous page)

```
    unsigned char input[CIPHER_LEN +TAG_LEN] = {...};
    unsigned char output[DATA_LEN] = {0};

    struct smw_aead_args args = {0};
    struct smw_aead_data_args data_args = {0};
    struct smw_aead_aad_args aad_args = {0};
    struct smw_aead_final_args final_args = {0};
    struct smw_aead_init_args init_args = {0};
    struct smw_key_descriptor key_desc = {0};
    struct smw_op_context *op_ctx = 0;

    init_args.subsystem_name = "TEE";
    init_args.operation_name = "DECRYPT";
    init_args.mode_name = "GCM";
    init_args.aad_length = AAD_LEN;
    init_args.tag_length = TAG_LEN;
    init_args.plaintext_length = DATA_LEN;

    // Allocate memory to pointer to operation context
    op_ctx = calloc(1, sizeof(*op_ctx));
    init_args.context = op_ctx;
    init_args.key_desc = &key_desc;

    // Initialize multi-part AEAD operation
    res = smw_aead_init(&init_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    // Add additional data to an active AEAD operation.
    aad_args.aad = aad;
    aad_args.aad_length = AAD_LEN;
    aad_args.context = init_args.context;
    res = smw_aead_update_add(&aad_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    /**
     * Decrypt 1st message fragment in an active
     * multi-part AEAD decryption operation.
     */
    data_args.input_length = 16;
    data_args.input = input;
    data_args.output_length = 16;
    data_args.output = output;
    data_args.context = init_args.context;
    res = smw_aead_update(&data_args);
    if (res != SMW_STATUS_OK)
        goto exit;
```

(continues on next page)

(continued from previous page)

```
    /**
     * Decrypt 2nd message fragment in an active
     * multi-part AEAD decryption operation.
     */
    data_args.input_length = 16;
    data_args.input = &input[16];
    data_args.output_length = 16;
    data_args.output = &output[16];
    data_args.context = init_args.context;
    res = smw_aead_update(&data_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    /**
     * Finish authenticating and decrypting the message
     * in an active multi-part AEAD operation.
     */
    final_args.operation_name = "DECRYPT";
    final_args.data.context = init_args.context;
    // Pass the tag
    final_args.data.input = &input[32];
    final_args.data.input_length = TAG_LEN;
    final_args.data.output = NULL;
    final_args.data.output_length = 0;
    final_args.tag_length = TAG_LEN;
    res = smw_aead_final(&final_args);
    if (res != SMW_STATUS_OK)
        goto exit;

    exit:
    if (op_ctx)
        free(op_ctx);

    return res;
}
```

3.2 PSA APIs

3.2.1 Cryptography APIs

3.2.1.1 Values

3.2.1.1.1 Introduction

This file declares macros to build and analyze values of integral types defined in crypto_types.h.

3.2.1.1.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.1.1.3 macro PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG

PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg)

An AEAD algorithm with the default tag length.

Parameters

- **aead_alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(aead_alg) is true).

3.2.1.1.3.1 Description

This macro can be used to construct the AEAD algorithm with default tag length from an AEAD algorithm with a shortened tag. See also *PSA_ALG_AEAD_WITH_SHORTENED_TAG()*.

3.2.1.1.3.2 Return

The corresponding AEAD algorithm with the default tag length for that algorithm.

3.2.1.1.4 macro PSA_ALG_AEAD_WITH_SHORTENED_TAG

PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length)

Macro to build a AEAD algorithm with a shortened tag.

Parameters

- **aead_alg** – An AEAD algorithm identifier (value of *typedef psa_algorithm_t* such that PSA_ALG_IS_AEAD(aead_alg) is true).
- **tag_length** – Desired length of the authentication tag in bytes.

3.2.1.1.4.1 Description

An AEAD algorithm with a shortened tag is similar to the corresponding AEAD algorithm, but has an authentication tag that consists of fewer bytes. Depending on the algorithm, the tag length might affect the calculation of the ciphertext.

The AEAD algorithm with a default length tag can be recovered using *PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()*.

3.2.1.1.4.2 Return

The corresponding AEAD algorithm with the specified tag length.

Unspecified if `aead_alg` is not a supported AEAD algorithm or if `tag_length` is not valid for the specified AEAD algorithm.

3.2.1.1.5 macro PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG

PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG(aead_alg, min_tag_length)

Macro to build an AEAD minimum-tag-length wildcard algorithm.

Parameters

- **aead_alg** – An AEAD algorithm: a value of *typedef psa_algorithm_t* such that PSA_ALG_IS_AEAD(aead_alg) is true.
- **min_tag_length** – Desired minimum length of the authentication tag in bytes. This must be at least 1 and at most the largest allowed tag length of the algorithm.

3.2.1.1.5.1 Description

A key with a minimum-tag-length AEAD wildcard algorithm as permitted algorithm policy can be used with all AEAD algorithms sharing the same base algorithm, and where the tag length of the specific algorithm is equal to or larger then the minimum tag length specified by the wildcard algorithm.

Note:

When setting the minimum required tag length to less than the smallest tag length allowed by the base algorithm, this effectively becomes an 'any-tag-length-allowed' policy for that base algorithm.

The AEAD algorithm with a default length tag can be recovered using *PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()*.

Compatible key types:

The resulting wildcard AEAD algorithm is compatible with the same key types as the AEAD algorithm used to construct it.

3.2.1.1.5.2 Return

The corresponding AEAD wildcard algorithm with the specified minimum tag length.

Unspecified if aead_alg is not a supported AEAD algorithm or if min_tag_length is less than 1 or too large for the specified AEAD algorithm.

3.2.1.1.6 macro PSA_ALG_DETERMINISTIC_ECDSA

`PSA_ALG_DETERMINISTIC_ECDSA`(hash_alg)

Deterministic ECDSA signature scheme, with hashing.

Parameters

- **`hash_alg`** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

3.2.1.1.6.1 Description

This algorithm can be used with both the message and hash signature functions.

Note:

When based on the same hash algorithm, the verification operations for PSA_ALG_ECDSA and PSA_ALG_DETERMINISTIC_ECDSA are identical. A signature created using PSA_ALG_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA. Similarly, a signature created using PSA_ALG_DETERMINISTIC_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA.

In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

This is the deterministic ECDSA signature scheme defined by Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) [RFC6979].

The representation of a signature is the same as with *`PSA_ALG_ECDSA()`*.

3.2.1.1.6.2 Return

The corresponding deterministic ECDSA signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.7 macro PSA_ALG_ECDSA

`PSA_ALG_ECDSA`(hash_alg)

The randomized ECDSA signature scheme, with hashing.

Parameters

- **`hash_alg`** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

3.2.1.1.7.1 Description

This algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

Note:

When based on the same hash algorithm, the verification operations for PSA_ALG_ECDSA and PSA_ALG_DETERMINISTIC_ECDSA are identical. A signature created using PSA_ALG_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA. Similarly, a signature created using PSA_ALG_DETERMINISTIC_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA.

In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

This signature scheme is defined by SEC 1: Elliptic Curve Cryptography [SEC1], and also by Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA) [X9-62], with a random per-message secret number k.

The representation of the signature as a byte string consists of the concatenation of the signature values r and s. Each of r and s is encoded as an N-octet string, where N is the length of the base point of the curve in octets. Each value is represented in big-endian order, with the most significant octet first.

3.2.1.1.7.2 Return

The corresponding randomized ECDSA signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.8 macro PSA_ALG_FULL_LENGTH_MAC

`PSA_ALG_FULL_LENGTH_MAC`(mac_alg)

Macro to construct the MAC algorithm with a full length MAC, from a truncated MAC algorithm.

Parameters

- **`mac_alg`** – A MAC algorithm identifier (value of *`typedef psa_algorithm_t`* such that PSA_ALG_IS_MAC(mac_alg) is true). This can be a truncated or untruncated MAC algorithm.

3.2.1.1.8.1 Return

The corresponding MAC algorithm with a full length MAC.

Unspecified if alg is not a supported MAC algorithm.

3.2.1.1.9 macro PSA_ALG_AT_LEAST_THIS_LENGTH_MAC

PSA_ALG_AT_LEAST_THIS_LENGTH_MAC(mac_alg, min_mac_length)

Macro to build a MAC minimum-MAC-length wildcard algorithm.

Parameters

- **mac_alg** – A MAC algorithm: a value of *typedef psa_algorithm_t* such that PSA_ALG_IS_MAC(alg) is true. This can be a truncated or untruncated MAC algorithm.
- **min_mac_length** – Desired minimum length of the message authentication code inbytes. This must be at most the untruncated length of the MAC andmust be at least 1.

3.2.1.1.9.1 Description

A key with a minimum-MAC-length MAC wildcard algorithm as permitted algorithm policy can be used with all MAC algorithms sharing the same base algorithm, and where the (potentially truncated) MAC length of the specific algorithm is equal to or larger then the wildcard algorithm's minimum MAC length.

Note:

When setting the minimum required MAC length to less than the smallest MAC length allowed by the base algorithm, this effectively becomes an 'any-MAC-length-allowed' policy for that base algorithm.

The untruncated MAC algorithm can be recovered using *PSA_ALG_FULL_LENGTH_MAC()*.

Compatible key types:

The resulting wildcard MAC algorithm is compatible with the same key types as the MAC algorithm used to construct it.

3.2.1.1.9.2 Return

The corresponding MAC wildcard algorithm with the specified minimum MAC length.

Unspecified if mac_alg is not a supported MAC algorithm or if min_mac_length is less than 1 or too large for the specified MAC algorithm.

3.2.1.1.10 macro PSA_ALG_GET_HASH

PSA_ALG_GET_HASH(alg)

Get the hash used by a composite algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.10.1 Description

The following composite algorithms require a hash algorithm:

- *PSA_ALG_ECDSA()*
- *PSA_ALG_HKDF()*
- *PSA_ALG_HMAC()*
- *PSA_ALG_RSA_OAEP()*
- *PSA_ALG_IS_RSA_PKCS1V15_SIGN()*
- *PSA_ALG_RSA_PSS()*
- *PSA_ALG_TLS12_PRF()*
- *PSA_ALG_TLS12_PSK_TO_MS()*

3.2.1.1.10.2 Return

The underlying hash algorithm if `alg` is a composite algorithm that uses a hash algorithm.

PSA_ALG_NONE if `alg` is not a composite algorithm that uses a hash.

3.2.1.1.11 macro PSA_ALG_HKDF

PSA_ALG_HKDF(hash_alg)

Macro to build an HKDF algorithm.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true).

3.2.1.1.11.1 Description

This is the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) specified by HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [RFC5869].

This key derivation algorithm uses the following inputs:

- PSA_KEY_DERIVATION_INPUT_SALT is the salt used in the "extract" step. It is optional; if omitted, the derivation uses an empty salt.
- PSA_KEY_DERIVATION_INPUT_SECRET is the secret key used in the "extract" step.
- PSA_KEY_DERIVATION_INPUT_INFO is the info string used in the "expand" step.

If PSA_KEY_DERIVATION_INPUT_SALT is provided, it must be before PSA_KEY_DERIVATION_INPUT_SECRET. PSA_KEY_DERIVATION_INPUT_INFO can be provided at any time after setup and before starting to generate output.

Each input may only be passed once.

3.2.1.1.11.2 Return

The corresponding HKDF algorithm. For example, PSA_ALG_HKDF(PSA_ALG_SHA_256) is HKDF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.12 macro PSA_ALG_HMAC

PSA_ALG_HMAC(hash_alg)

Macro to build an HMAC message-authentication-code algorithm from an underlying hash algorithm.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true).

3.2.1.1.12.1 Description

For example, PSA_ALG_HMAC(PSA_ALG_SHA_256) is HMAC-SHA-256.

The HMAC construction is defined in HMAC: Keyed-Hashing for Message Authentication [RFC2104].

3.2.1.1.12.2 Return

The corresponding HMAC algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.13 macro PSA_ALG_IS_AEAD

PSA_ALG_IS_AEAD(alg)

Whether the specified algorithm is an authenticated encryption with associated data (AEAD) algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.13.1 Return

1 if `alg` is an AEAD algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.14 macro PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER

PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg)

Whether the specified algorithm is an AEAD mode on a block cipher.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.14.1 Return

1 if alg is an AEAD algorithm which is an AEAD mode based on a block cipher, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.15 macro PSA_ALG_IS_ASYMMETRIC_ENCRYPTION

PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg)

Whether the specified algorithm is an asymmetric encryption algorithm, also known as public-key encryption algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.15.1 Return

1 if alg is an asymmetric encryption algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.16 macro PSA_ALG_IS_BLOCK_CIPHER_MAC

PSA_ALG_IS_BLOCK_CIPHER_MAC(alg)

Whether the specified algorithm is a MAC algorithm based on a block cipher.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.16.1 Return

1 if alg is a MAC algorithm based on a block cipher, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.17 macro PSA_ALG_IS_CIPHER

PSA_ALG_IS_CIPHER(alg)

Whether the specified algorithm is a symmetric cipher algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.17.1 Return

1 if `alg` is a symmetric cipher algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.18 macro PSA_ALG_IS_DETERMINISTIC_ECDSA

PSA_ALG_IS_DETERMINISTIC_ECDSA(alg)

Whether the specified algorithm is deterministic ECDSA.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.18.1 Description

See also *PSA_ALG_IS_ECDSA()* and *PSA_ALG_IS_RANDOMIZED_ECDSA()*.

3.2.1.1.18.2 Return

1 if `alg` is a deterministic ECDSA algorithm, 0 otherwise.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.19 macro PSA_ALG_IS_ECDH

PSA_ALG_IS_ECDH(alg)

Whether the specified algorithm is an elliptic curve Diffie-Hellman algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.19.1 Description

This includes the raw elliptic curve Diffie-Hellman algorithm as well as elliptic curve Diffie-Hellman followed by any supporter key derivation algorithm.

3.2.1.1.19.2 Return

1 if `alg` is an elliptic curve Diffie-Hellman algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier.

3.2.1.1.20 macro PSA_ALG_IS_ECDSA

PSA_ALG_IS_ECDSA(alg)

Whether the specified algorithm is ECDSA.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.20.1 Return

1 if `alg` is an ECDSA algorithm, 0 otherwise.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.21 macro PSA_ALG_IS_FFDH

PSA_ALG_IS_FFDH(alg)

Whether the specified algorithm is a finite field Diffie-Hellman algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.21.1 Description

This includes the raw finite field Diffie-Hellman algorithm as well as finite-field Diffie-Hellman followed by any supporter key derivation algorithm.

3.2.1.1.21.2 Return

1 if `alg` is a finite field Diffie-Hellman algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier.

3.2.1.1.22 macro PSA_ALG_IS_HASH

PSA_ALG_IS_HASH(alg)

Whether the specified algorithm is a hash algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.22.1 Description

See Hash algorithms for a list of defined hash algorithms.

3.2.1.1.22.2 Return

1 if `alg` is a hash algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.23 macro PSA_ALG_IS_HASH_AND_SIGN

PSA_ALG_IS_HASH_AND_SIGN(alg)

Whether the specified algorithm is a hash-and-sign algorithm that signs exactly the hash value.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.23.1 Description

This macro identifies algorithms that can be used with *psa_sign_hash()* that use the exact message hash value as an input the signature operation. This excludes hash-and-sign algorithms that require a encoded or modified hash for the signature step in the algorithm, such as PSA_ALG_RSA_PKCS1V15_SIGN_RAW.

3.2.1.1.23.2 Return

1 if `alg` is a hash-and-sign algorithm that signs exactly the hash value, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.24 macro PSA_ALG_IS_HASH_EDDSA

PSA_ALG_IS_HASH_EDDSA(alg)

Whether the specified algorithm is HashEdDSA.

Parameters

- **alg** – An algorithm identifier: a value of *typedef psa_algorithm_t*.

3.2.1.1.24.1 Return

1 if `alg` is a HashEdDSA algorithm, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.25 macro PSA_ALG_IS_HKDF

PSA_ALG_IS_HKDF(alg)

Whether the specified algorithm is an HKDF algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.25.1 Description

HKDF is a family of key derivation algorithms that are based on a hash function and the HMAC construction.

3.2.1.1.25.2 Return

1 if `alg` is an HKDF algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier.

3.2.1.1.26 macro PSA_ALG_IS_HMAC

PSA_ALG_IS_HMAC(alg)

Whether the specified algorithm is an HMAC algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.26.1 Description

HMAC is a family of MAC algorithms that are based on a hash function.

3.2.1.1.26.2 Return

1 if alg is an HMAC algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.27 macro PSA_ALG_IS_KEY_AGREEMENT

PSA_ALG_IS_KEY_AGREEMENT(alg)

Whether the specified algorithm is a key agreement algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.27.1 Return

1 if `alg` is a key agreement algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.28 macro PSA_ALG_IS_KEY_DERIVATION

PSA_ALG_IS_KEY_DERIVATION(alg)

Whether the specified algorithm is a key derivation algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.28.1 Return

1 if `alg` is a key derivation algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.29 macro PSA_ALG_IS_KEY_DERIVATION_STRETCHING

PSA_ALG_IS_KEY_DERIVATION_STRETCHING(alg)

Whether the specified algorithm is a key-stretching or password-hashing algorithm.

Parameters

- **alg** – An algorithm identifier: a value of *typedef psa_algorithm_t*.

3.2.1.1.29.1 Description

A key-stretching or password-hashing algorithm is a key derivation algorithm that is suitable for use with a low-entropy secret such as a password. Equivalently, it's a key derivation algorithm that uses a PSA_KEY_DERIVATION_INPUT_PASSWORD input step.

3.2.1.1.29.2 Return

1 if alg is a key-stretching or password-hashing algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported key derivation algorithm identifier.

3.2.1.1.30 macro PSA_ALG_IS_MAC

PSA_ALG_IS_MAC(alg)

Whether the specified algorithm is a MAC algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.30.1 Return

1 if `alg` is a MAC algorithm, 0 otherwise.

3.2.1.1.31 macro PSA_ALG_IS_MAC_TRUNCATED

PSA_ALG_IS_MAC_TRUNCATED(alg)

Whether the specified algorithm is a MAC truncated algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.31.1 Return

1 if `alg` is a MAC truncated algorithm, 0 otherwise.

3.2.1.1.32 macro PSA_ALG_IS_PBKDF2_HMAC

PSA_ALG_IS_PBKDF2_HMAC(alg)

Whether the specified algorithm is a PBKDF2-HMAC algorithm.

Parameters

- **alg** – An algorithm identifier: a value of *typedef psa_algorithm_t*.

3.2.1.1.32.1 Return

1 if alg is a PBKDF2-HMAC algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported key derivation algorithm identifier.

3.2.1.1.33 macro PSA_ALG_IS_RANDOMIZED_ECDSA

PSA_ALG_IS_RANDOMIZED_ECDSA(alg)

Whether the specified algorithm is randomized ECDSA.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.33.1 Description

See also *PSA_ALG_IS_ECDSA()* and *PSA_ALG_IS_DETERMINISTIC_ECDSA()*.

3.2.1.1.33.2 Return

1 if `alg` is a randomized ECDSA algorithm, 0 otherwise.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.34 macro PSA_ALG_IS_RAW_KEY_AGREEMENT

PSA_ALG_IS_RAW_KEY_AGREEMENT(alg)

Whether the specified algorithm is a raw key agreement algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.34.1 Description

A raw key agreement algorithm is one that does not specify a key derivation function. Usually, raw key agreement algorithms are constructed directly with a PSA_ALG_xxx macro while non-raw key agreement algorithms are constructed with *PSA_ALG_KEY_AGREEMENT()*.

The raw key agreement algorithm can be extracted from a full key agreement algorithm identifier using *PSA_ALG_KEY_AGREEMENT_GET_BASE()*.

3.2.1.1.34.2 Return

1 if `alg` is a raw key agreement algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.35 macro PSA_ALG_IS_RSA_OAEP

PSA_ALG_IS_RSA_OAEP(alg)

Whether the specified algorithm is an RSA OAEP encryption algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.35.1 Return

1 if `alg` is an RSA OAEP algorithm, 0 otherwise.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.36 macro PSA_ALG_IS_RSA_PKCS1V15_SIGN

PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg)

Whether the specified algorithm is an RSA PKCS#1 v1.5 signature algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.36.1 Return

1 if `alg` is an RSA PKCS#1 v1.5 signature algorithm, 0 otherwise.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.37 macro PSA_ALG_IS_RSA_PSS

PSA_ALG_IS_RSA_PSS(alg)

Whether the specified algorithm is an RSA PSS signature algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.37.1 Return

1 if alg is an RSA PSS signature algorithm, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.38 macro PSA_ALG_IS_RSA_PSS_ANY_SALT

PSA_ALG_IS_RSA_PSS_ANY_SALT(alg)

Whether the specified algorithm is an RSA PSS signature algorithm that permits any salt length.

Parameters

- **alg** – An algorithm identifier: a value of *typedef psa_algorithm_t*.

3.2.1.1.38.1 Description

An RSA PSS signature algorithm that permits any salt length is constructed using *PSA_ALG_RSA_PSS_ANY_SALT()*. See also *PSA_ALG_IS_RSA_PSS()* and *PSA_ALG_IS_RSA_PSS_STANDARD_SALT()*

3.2.1.1.38.2 Return

1 if alg is an RSA PSS signature algorithm that permits any salt length, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.39 macro PSA_ALG_IS_RSA_PSS_STANDARD_SALT

PSA_ALG_IS_RSA_PSS_STANDARD_SALT(alg)

Whether the specified algorithm is an RSA PSS signature algorithm that requires the standard salt length.

Parameters

- **alg** – An algorithm identifier: a value of *typedef psa_algorithm_t*.

3.2.1.1.39.1 Description

An RSA PSS signature algorithm that requires the standard salt length is constructed using *PSA_ALG_RSA_PSS()*.

See also *PSA_ALG_IS_RSA_PSS()* and *PSA_ALG_IS_RSA_PSS_ANY_SALT()*.

3.2.1.1.39.2 Return

1 if alg is an RSA PSS signature algorithm that requires the standard salt length, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

3.2.1.1.40 macro PSA_ALG_IS_SIGN

PSA_ALG_IS_SIGN(alg)

Whether the specified algorithm is an asymmetric signature algorithm, also known as public-key signature algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.40.1 Return

1 if `alg` is an asymmetric signature algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.41 macro PSA_ALG_IS_SIGN_HASH

PSA_ALG_IS_SIGN_HASH(alg)

Whether the specified algorithm is a signature algorithm that can be used with *psa_sign_hash()* and *psa_verify_hash()*.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.41.1 Return

1 if `alg` is a signature algorithm that can be used to sign a hash. 0 `alg` alg is a signature algorithm that can only be used to sign a message. 0 if `alg` is not a signature algorithm. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.42 macro PSA_ALG_IS_SIGN_MESSAGE

PSA_ALG_IS_SIGN_MESSAGE(alg)

Whether the specified algorithm is a signature algorithm that can be used with *psa_sign_message()* and *psa_verify_message()*.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.42.1 Return

1 if `alg` is a signature algorithm that can be used to sign a message. 0 if `alg` is a signature algorithm that can only be used to sign an already-calculated hash. 0 if `alg` is not a signature algorithm. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.43 macro PSA_ALG_IS_STREAM_CIPHER

PSA_ALG_IS_STREAM_CIPHER(alg)

Whether the specified algorithm is a stream cipher.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.43.1 Description

A stream cipher is a symmetric cipher that encrypts or decrypts messages by applying a bitwise-xor with a stream of bytes that is generated from a key.

3.2.1.1.43.2 Return

1 if `alg` is a stream cipher algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier or if it is not a symmetric cipher algorithm.

3.2.1.1.44 macro PSA_ALG_IS_TLS12_PRF

PSA_ALG_IS_TLS12_PRF(alg)

Whether the specified algorithm is a TLS-1.2 PRF algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.44.1 Return

1 if `alg` is a TLS-1.2 PRF algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier.

3.2.1.1.45 macro PSA_ALG_IS_TLS12_PSK_TO_MS

PSA_ALG_IS_TLS12_PSK_TO_MS(alg)

Whether the specified algorithm is a TLS-1.2 PSK to MS algorithm.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.45.1 Return

1 if `alg` is a TLS-1.2 PSK to MS algorithm, 0 otherwise. This macro can return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier.

3.2.1.1.46 macro PSA_ALG_IS_WILDCARD

PSA_ALG_IS_WILDCARD(alg)

Whether the specified algorithm encoding is a wildcard.

Parameters

- **alg** – An algorithm identifier (value of *typedef psa_algorithm_t*).

3.2.1.1.46.1 Description

Wildcard algorithm values can only be used to set the permitted algorithm field in a key policy, wildcard values cannot be used to perform an operation.

See PSA_ALG_ANY_HASH for example of how a wildcard algorithm can be used in a key policy.

3.2.1.1.46.2 Return

1 if `alg` is a wildcard algorithm encoding.

0 if `alg` is a non-wildcard algorithm encoding that is suitable for an operation.

This macro can return either 0 or 1 if `alg` is not a supported algorithm identifier.

3.2.1.1.47 macro PSA_ALG_KEY_AGREEMENT

PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg)

Macro to build a combined algorithm that chains a key agreement with a key derivation.

Parameters

- **ka_alg** – A key agreement algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_AGREEMENT(ka_alg) is true).
- **kdf_alg** – A key derivation algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_DERIVATION(kdf_alg) is true).

3.2.1.1.47.1 Description

A combined key agreement algorithm is used with a multi-part key derivation operation, using a call to *psa_key_derivation_key_agreement()*.

The component parts of a key agreement algorithm can be extracted using *PSA_ALG_KEY_AGREEMENT_GET_BASE()* and *PSA_ALG_KEY_AGREEMENT_GET_KDF()*.

3.2.1.1.47.2 Return

The corresponding key agreement and derivation algorithm.

Unspecified if `ka_alg` is not a supported key agreement algorithm or `kdf_alg` is not a supported key derivation algorithm.

3.2.1.1.48 macro PSA_ALG_KEY_AGREEMENT_GET_BASE

PSA_ALG_KEY_AGREEMENT_GET_BASE(alg)

Get the raw key agreement algorithm from a full key agreement algorithm.

Parameters

- **alg** – A key agreement algorithm identifier (value of *typedef psa_algorithm_t* such that PSA_ALG_IS_KEY_AGREEMENT(alg) is true).

3.2.1.1.48.1 Description

See also *PSA_ALG_KEY_AGREEMENT()* and *PSA_ALG_KEY_AGREEMENT_GET_KDF()*.

3.2.1.1.48.2 Return

The underlying raw key agreement algorithm if `alg` is a key agreement algorithm.

Unspecified if `alg` is not a key agreement algorithm or if it is not supported by the implementation.

3.2.1.1.49 macro PSA_ALG_KEY_AGREEMENT_GET_KDF

PSA_ALG_KEY_AGREEMENT_GET_KDF(alg)

Get the key derivation algorithm used in a full key agreement algorithm.

Parameters

- **alg** – A key agreement algorithm identifier (value of *typedef psa_algorithm_t* such that PSA_ALG_IS_KEY_AGREEMENT(alg) is true).

3.2.1.1.49.1 Description

See also *PSA_ALG_KEY_AGREEMENT()* and *PSA_ALG_KEY_AGREEMENT_GET_BASE()*.

3.2.1.1.49.2 Return

The underlying key derivation algorithm if `alg` is a key agreement algorithm.

Unspecified if `alg` is not a key agreement algorithm or if it is not supported by the implementation.

3.2.1.1.50 macro PSA_ALG_PBKDF2_HMAC

PSA_ALG_PBKDF2_HMAC(hash_alg)

Macro to build a PBKDF2-HMAC password-hashing or key-stretching algorithm.

Parameters

- **hash_alg** – A hash algorithm: a value of *typedef psa_algorithm_t* such that PSA_ALG_IS_HASH(hash_alg) is true.

3.2.1.1.50.1 Description

PBKDF2 is specified by PKCS #5: Password-Based Cryptography Specification Version 2.1 [RFC8018] §5.2. This macro constructs a PBKDF2 algorithm that uses a pseudo-random function based on HMAC with the specified hash. This key derivation algorithm uses the following inputs, which must be provided in the following order:

- PSA_KEY_DERIVATION_INPUT_COST is the iteration count. This input step must be used exactly once.
- PSA_KEY_DERIVATION_INPUT_SALT is the salt. This input step must be used one or more times; if used several times, the inputs will be concatenated. This can be used to build the final salt from multiple sources, both public and secret (also known as pepper).

- PSA_KEY_DERIVATION_INPUT_PASSWORD is the password to be hashed. This input step must be used exactly once.

Compatible key types:

- PSA_KEY_TYPE_DERIVE (for password input)
- PSA_KEY_TYPE_PASSWORD (for password input)
- PSA_KEY_TYPE_PEPPER (for salt input)
- PSA_KEY_TYPE_RAW_DATA (for salt input)
- PSA_KEY_TYPE_PASSWORD_HASH (for key verification)

3.2.1.1.50.2 Return

The corresponding PBKDF2-HMAC-XXX algorithm. For example, PSA_ALG_PBKDF2_HMAC(PSA_ALG_SHA_256) is the algorithm identifier for PBKDF2-HMAC-SHA-256.

Unspecified if hash_alg is not a supported hash algorithm.

3.2.1.1.51 macro PSA_ALG_RSA_OAEP

`PSA_ALG_RSA_OAEP`(hash_alg)

The RSA OAEP asymmetric encryption algorithm.

Parameters

- **`hash_alg`** – The hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true) to use for MGF1.

3.2.1.1.51.1 Description

This encryption scheme is defined by [RFC8017] §7.1 under the name RSAES-OAEP, with the mask generation function MGF1 defined in [RFC8017] Appendix B.

3.2.1.1.51.2 Return

The corresponding RSA OAEP encryption algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.52 macro PSA_ALG_RSA_PKCS1V15_SIGN

PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg)

The RSA PKCS#1 v1.5 message signature scheme, with hashing.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

3.2.1.1.52.1 Description

This algorithm can be used with both the message and hash signature functions.

This signature scheme is defined by PKCS #1: RSA Cryptography Specifications Version 2.2 [RFC8017] §8.2 under the name RSASSA-PKCS1-v1_5.

When used with *psa_sign_hash()* or *psa_verify_hash()*, the provided hash parameter is used as H from step 2 onwards in the message encoding algorithm EMSA-PKCS1-V1_5-ENCODE() in [RFC8017] §9.2. H is usually the message digest, using the `hash_alg` hash algorithm.

3.2.1.1.52.2 Return

The corresponding RSA PKCS#1 v1.5 signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.53 macro PSA_ALG_RSA_PSS

PSA_ALG_RSA_PSS(hash_alg)

The RSA PSS message signature scheme, with hashing.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

3.2.1.1.53.1 Description

This algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the signature scheme defined by [RFC8017] §8.1 under the name RSASSA-PSS, with the following options:

- The mask generation function is MGF1 defined by [RFC8017] Appendix B.
- The salt length is equal to the length of the hash.

- The specified hash algorithm is used to hash the input message, to create the salted hash, and for the mask generation.

3.2.1.1.53.2 Return

The corresponding RSA PSS signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.54 macro PSA_ALG_RSA_PSS_ANY_SALT

PSA_ALG_RSA_PSS_ANY_SALT(hash_alg)

The RSA PSS message signature scheme, with hashing. This variant permits any salt length for signature verification.

Parameters

- **hash_alg** – A hash algorithm: a value of *typedef psa_algorithm_t* such that PSA_ALG_IS_HASH(hash_alg) is true. This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

3.2.1.1.54.1 Description

This algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the signature scheme defined by [RFC8017] §8.1 under the name RSASSA-PSS, with the following options:

- The mask generation function is MGF1 defined by [RFC8017] Appendix B.
- When creating a signature, the salt length is equal to the length of the hash, or the largest possible salt length for the algorithm and key size if that is smaller than the hash length.
- When verifying a signature, any salt length permitted by the RSASSA-PSS signature algorithm is accepted.
- The specified hash algorithm is used to hash the input message, to create the salted hash, and for the mask generation.

Note:

The *PSA_ALG_RSA_PSS()* algorithm is equivalent to *PSA_ALG_RSA_PSS_ANY_SALT()* when creating a signature, but is strict about the permitted salt length when verifying a signature.

Compatible key types:

- PSA_KEY_TYPE_RSA_KEY_PAIR
- PSA_KEY_TYPE_RSA_PUBLIC_KEY (signature verification only)

3.2.1.1.54.2 Return

The corresponding RSA PSS signature algorithm.

Unspecified if hash_alg is not a supported hash algorithm.

3.2.1.1.55 macro PSA_ALG_TLS12_PRF

PSA_ALG_TLS12_PRF(hash_alg)

Macro to build a TLS-1.2 PRF algorithm.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true).

3.2.1.1.55.1 Description

TLS 1.2 uses a custom pseudorandom function (PRF) for key schedule, specified in The Transport Layer Security (TLS) Protocol Version 1.2 [RFC5246] §5. It is based on HMAC and can be used with either SHA-256 or SHA-384.

This key derivation algorithm uses the following inputs, which must be passed in the order given here:

- PSA_KEY_DERIVATION_INPUT_SEED is the seed.
- PSA_KEY_DERIVATION_INPUT_SECRET is the secret key.
- PSA_KEY_DERIVATION_INPUT_LABEL is the label.

Each input may only be passed once.

For the application to TLS-1.2 key expansion:

- The seed is the concatenation of ServerHello.Random + ClientHello.Random.
- The label is "key expansion".

3.2.1.1.55.2 Return

The corresponding TLS-1.2 PRF algorithm. For example, PSA_ALG_TLS12_PRF(PSA_ALG_SHA_256) represents the TLS 1.2 PRF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.56 macro PSA_ALG_TLS12_PSK_TO_MS

PSA_ALG_TLS12_PSK_TO_MS(hash_alg)

Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

Parameters

- **hash_alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(hash_alg) is true).

3.2.1.1.56.1 Description

In a pure-PSK handshake in TLS 1.2, the master secret (MS) is derived from the pre-shared key (PSK) through the application of padding (Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) [RFC4279] §2) and the TLS-1.2 PRF (The Transport Layer Security (TLS) Protocol Version 1.2 [RFC5246] §5). The latter is based on HMAC and can be used with either SHA-256 or SHA-384.

This key derivation algorithm uses the following inputs, which must be passed in the order given here:

- PSA_KEY_DERIVATION_INPUT_SEED is the seed.
- PSA_KEY_DERIVATION_INPUT_SECRET is the PSK. The PSK must not be larger than PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE.
- PSA_KEY_DERIVATION_INPUT_LABEL is the label.

Each input may only be passed once.

For the application to TLS-1.2:

- The seed, which is forwarded to the TLS-1.2 PRF, is the concatenation of the ClientHello.Random + ServerHello.Random.
- The label is "master secret" or "extended master secret".

3.2.1.1.56.2 Return

The corresponding TLS-1.2 PSK to MS algorithm. For example, PSA_ALG_TLS12_PSK_TO_MS(PSA_ALG_SHA_256) represents the TLS-1.2 PSK to MasterSecret derivation PRF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

3.2.1.1.57 macro PSA_ALG_TRUNCATED_MAC

PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length)

Macro to build a truncated MAC algorithm.

Parameters

- **mac_alg** – A MAC algorithm identifier (value of *typedef psa_algorithm_t* such that PSA_ALG_IS_MAC(mac_alg) is true). This can be a truncated or untruncated MAC algorithm.

- **mac_length** – Desired length of the truncated MAC in bytes. This must be at most the full length of the MAC and must be at least an implementation-specified minimum. The implementation-specified minimum must not be zero.

3.2.1.1.57.1 Description

A truncated MAC algorithm is identical to the corresponding MAC algorithm except that the MAC value for the truncated algorithm consists of only the first `mac_length` bytes of the MAC value for the untruncated algorithm.

Note:

This macro might allow constructing algorithm identifiers that are not valid, either because the specified length is larger than the untruncated MAC or because the specified length is smaller than permitted by the implementation.

Note:

It is implementation-defined whether a truncated MAC that is truncated to the same length as the MAC of the untruncated algorithm is considered identical to the untruncated algorithm for policy comparison purposes.

The full-length MAC algorithm can be recovered using *PSA_ALG_FULL_LENGTH_MAC()*.

3.2.1.1.57.2 Return

The corresponding MAC algorithm with the specified length.

Unspecified if alg is not a supported MAC algorithm or if `mac_length` is too small or too large for the specified MAC algorithm.

3.2.1.1.58 macro PSA_BLOCK_CIPHER_BLOCK_LENGTH

PSA_BLOCK_CIPHER_BLOCK_LENGTH(type)

The block size of a block cipher.

Parameters

- **type** – A cipher key type (value of *typedef psa_key_type_t*).

3.2.1.1.58.1 Description

Note:

It is possible to build stream cipher algorithms on top of a block cipher, for example CTR mode (PSA_ALG_CTR). This macro only takes the key type into account, so it cannot be used to determine the size of the data that *psa_cipher_update()* might buffer for future processing in general.

Note:

This macro expression is a compile-time constant if `type` is a compile-time constant.

Warning:

This macro is permitted to evaluate its argument multiple times.

See also PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE.

3.2.1.1.58.2 Return

The block size for a block cipher, or 1 for a stream cipher. The return value is undefined if `type` is not a supported cipher key type.

3.2.1.1.59 PSA_DH_FAMILY_RFC7919

Finite-field Diffie-Hellman groups defined for TLS in RFC 7919.

This family includes groups with the following key sizes (in bits): 2048, 3072, 4096, 6144, 8192. An implementation can support all of these sizes or only a subset.

Keys is this group can only be used with the PSA_ALG_FFDH key agreement algorithm.

These groups are defined by Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS) [RFC7919] Appendix A.

3.2.1.1.60 PSA_ECC_FAMILY_BRAINPOOL_P_R1

Brainpool P random curves.

This family comprises the following curves:

- brainpoolP160r1 : key_bits = 160 (Deprecated)
- brainpoolP192r1 : key_bits = 192
- brainpoolP224r1 : key_bits = 224
- brainpoolP256r1 : key_bits = 256
- brainpoolP320r1 : key_bits = 320
- brainpoolP384r1 : key_bits = 384
- brainpoolP512r1 : key_bits = 512

They are defined in Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation [RFC5639].

Warning

The 160-bit curve brainpoolP160r1 is weak and deprecated and is only recommended for use in legacy protocols.

3.2.1.1.61 PSA_ECC_FAMILY_FRP

Curve used primarily in France and elsewhere in Europe.

This family comprises one 256-bit curve:

- FRP256v1 : key_bits = 256

This is defined by Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française [FRP].

3.2.1.1.62 PSA_ECC_FAMILY_MONTGOMERY

Montgomery curves.

This family comprises the following Montgomery curves:

- Curve25519 : key_bits = 255
- Curve448 : key_bits = 448

Keys in this family can only be used with the PSA_ALG_ECDH key agreement algorithm.

Curve25519 is defined in Curve25519: new Diffie-Hellman speed records [Curve25519]. Curve448 is defined in Ed448-Goldilocks, a new elliptic curve [Curve448].

3.2.1.1.63 PSA_ECC_FAMILY_SECP_K1

SEC Koblitz curves over prime fields.

This family comprises the following curves:

- secp192k1 : key_bits = 192
- secp224k1 : key_bits = 225
- secp256k1 : key_bits = 256

They are defined in SEC 2: Recommended Elliptic Curve Domain Parameters [SEC2].

3.2.1.1.64 PSA_ECC_FAMILY_SECP_R1

SEC random curves over prime fields.

This family comprises the following curves:

- secp192r1 : key_bits = 192
- secp224r1 : key_bits = 224
- secp256r1 : key_bits = 256
- secp384r1 : key_bits = 384
- secp521r1 : key_bits = 521

They are defined in [SEC2]

3.2.1.1.65 PSA_ECC_FAMILY_SECP_R2

Warning:
This family of curves is weak and deprecated.

This family comprises the following curves:

- secp160r2 : key_bits = 160 (Deprecated)

It is defined in the superseded SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0 [SEC2v1].

3.2.1.1.66 PSA_ECC_FAMILY_SECT_K1

SEC Koblitz curves over binary fields.

This family comprises the following curves:

- sect163k1 : key_bits = 163 (Deprecated)
- sect233k1 : key_bits = 233
- sect239k1 : key_bits = 239
- sect283k1 : key_bits = 283
- sect409k1 : key_bits = 409
- sect571k1 : key_bits = 571

They are defined in [SEC2].

Warning:
The 163-bit curve sect163k1 is weak and deprecated and is only recommended for use in legacy protocols.

3.2.1.1.67 PSA_ECC_FAMILY_SECT_R1

SEC random curves over binary fields.

This family comprises the following curves:

- sect163r1 : key_bits = 163 (Deprecated)
- sect233r1 : key_bits = 233
- sect283r1 : key_bits = 283
- sect409r1 : key_bits = 409
- sect571r1 : key_bits = 571

They are defined in [SEC2].

Warning:
The 163-bit curve sect163r1 is weak and deprecated and is only recommended for use in legacy protocols.

3.2.1.1.68 PSA_ECC_FAMILY_SECT_R2

SEC additional random curves over binary fields.

This family comprises the following curves:

- sect163r2 : key_bits = 163 (Deprecated)

It is defined in [SEC2].

Warning:
The 163-bit curve sect163r2 is weak and deprecated and is only recommended for use in legacy protocols.

3.2.1.1.69 PSA_ECC_FAMILY_TWISTED_EDWARDS

Twisted Edwards curves.

This family comprises the following twisted Edwards curves:

- Edwards25519 : key_bits = 255. This curve is birationally equivalent to Curve25519.
- Edwards448 : key_bits = 448. This curve is birationally equivalent to Curve448.

Edwards25519 is defined in Twisted Edwards curves [Ed25519]. Edwards448 is defined in Ed448-Goldilocks, a new elliptic curve [Curve448].

Compatible algorithms:

- PSA_ALG_PURE_EDDSA
- PSA_ALG_ED25519PH (Edwards25519 only)
- PSA_ALG_ED448PH (Edwards448 only)

3.2.1.1.70 PSA_KEY_DERIVATION_INPUT_CONTEXT

A context for key derivation.

Warning: Not supported

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

3.2.1.1.71 PSA_KEY_DERIVATION_INPUT_COST

A cost parameter for password hashing or key stretching.

Warning: Not supported

This must be a direct input, passed to *psa_key_derivation_input_integer()*.

3.2.1.1.72 PSA_KEY_DERIVATION_INPUT_INFO

An information string for key derivation.

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

3.2.1.1.73 PSA_KEY_DERIVATION_INPUT_LABEL

A label for key derivation.

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

3.2.1.1.74 PSA_KEY_DERIVATION_INPUT_PASSWORD

A low-entropy secret input for password hashing or key stretching.

Warning: Not supported

This is usually a key of type PSA_KEY_TYPE_PASSWORD passed to *psa_key_derivation_input_key()* or a direct input passed to *psa_key_derivation_input_bytes()* that is a password or passphrase. It can also be high-entropy secret, for example, a key of type PSA_KEY_TYPE_DERIVE, or the shared secret resulting from a key agreement.

If the secret is a direct input, the derivation operation cannot be used to derive keys: the operation will not allow a call to *psa_key_derivation_output_key()*.

3.2.1.1.75 PSA_KEY_DERIVATION_INPUT_SALT

A salt for key derivation.

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

3.2.1.1.76 PSA_KEY_DERIVATION_INPUT_SECRET

A secret input for key derivation.

This is typically a key of type PSA_KEY_TYPE_DERIVE passed to *psa_key_derivation_input_key()*, or the shared secret resulting from a key agreement obtained via *psa_key_derivation_key_agreement()*.

The secret can also be a direct input passed to *psa_key_derivation_input_bytes()*. In this case, the derivation operation cannot be used to derive keys: the operation will only allow *psa_key_derivation_output_bytes()*, not *psa_key_derivation_output_key()*.

3.2.1.1.77 PSA_KEY_DERIVATION_INPUT_SEED

A seed for key derivation.

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

3.2.1.1.78 PSA_KEY_ID_NULL

The null key identifier.

The null key identifier is always invalid, except when used without in a call to *psa_destroy_key()* which will return PSA_SUCCESS.

3.2.1.1.79 PSA_KEY_ID_USER_MAX

The maximum value for a key identifier chosen by the application.

3.2.1.1.80 PSA_KEY_ID_USER_MIN

The minimum value for a key identifier chosen by the application.

3.2.1.1.81 PSA_KEY_ID_VENDOR_MAX

The maximum value for a key identifier chosen by the implementation.

3.2.1.1.82 PSA_KEY_ID_VENDOR_MIN

The minimum value for a key identifier chosen by the implementation.

3.2.1.1.83 macro PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION

`PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION`(persistence, location)

Construct a lifetime from a persistence level and a location.

Parameters

- **persistence** – The persistence level (value of *typedef psa_key_persistence_t*).
- **location** – The location indicator (value of *typedef psa_key_location_t*).

3.2.1.1.83.1 Return

The constructed lifetime value.

3.2.1.1.84 macro PSA_KEY_LIFETIME_GET_LOCATION

PSA_KEY_LIFETIME_GET_LOCATION(lifetime)

Extract the location indicator from a key lifetime.

Parameters

- **lifetime** – The lifetime value to query (value of *typedef psa_key_lifetime_t*).

3.2.1.1.85 macro PSA_KEY_LIFETIME_GET_PERSISTENCE

PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime)

Extract the persistence level from a key lifetime.

Parameters

- **lifetime** – The lifetime value to query (value of *typedef psa_key_lifetime_t*).

3.2.1.1.86 macro PSA_KEY_LIFETIME_IS_VOLATILE

PSA_KEY_LIFETIME_IS_VOLATILE(lifetime)

Whether a key lifetime indicates that the key is volatile.

Parameters

- **lifetime** – The lifetime value to query (value of *typedef psa_key_lifetime_t*).

3.2.1.1.86.1 Description

A volatile key is automatically destroyed by the implementation when the application instance terminates. In particular, a volatile key is automatically destroyed on a power reset of the device.

A key that is not volatile is persistent. Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs,for example, a factory reset.

3.2.1.1.86.2 Return

1 if the key is volatile, otherwise 0.

3.2.1.1.87 PSA_KEY_LIFETIME_PERSISTENT

The default lifetime for persistent keys.

A persistent key remains in storage until it is explicitly destroyed or until the corresponding storage area is wiped. This specification does not define any mechanism to wipe a storage area. Implementations are permitted to provide their own mechanism, for example, to perform a factory reset, to prepare for device refurbishment, or to uninstall an application.

This lifetime value is the default storage area for the calling application. Implementations can offer other storage areas designated by other lifetime values as implementation-specific extensions.

3.2.1.1.88 PSA_KEY_LIFETIME_VOLATILE

The default lifetime for volatile keys.

A volatile key only exists as long as its identifier is not destroyed. The key material is guaranteed to be erased on a power reset.

A key with this lifetime is typically stored in the RAM area of the PSA Crypto subsystem. However this is an implementation choice. If an implementation stores data about the key in a non-volatile memory, it must release all the resources associated with the key and erase the key material if the calling application terminates.

3.2.1.1.89 PSA_KEY_LOCATION_LOCAL_STORAGE

The local storage area for persistent keys.

This storage area is available on all systems that can store persistent keys without delegating the storage to a third-party cryptoprocessor.

See *typedef psa_key_location_t* for more information.

3.2.1.1.90 PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT

The default secure element storage area for persistent keys.

This storage location is available on systems that have one or more secure elements that are able to store keys.

Vendor-defined locations must be provided by the system for storing keys in additional secure elements.

See *typedef psa_key_location_t* for more information.

3.2.1.1.91 PSA_KEY_PERSISTENCE_DEFAULT

The default persistence level for persistent keys.

See *typedef psa_key_persistence_t* for more information.

3.2.1.1.92 PSA_KEY_PERSISTENCE_READ_ONLY

A persistence level indicating that a key is never destroyed.

See *typedef psa_key_persistence_t* for more information.

3.2.1.1.93 PSA_KEY_PERSISTENCE_VOLATILE

The persistence level of volatile keys.

See *typedef psa_key_persistence_t* for more information.

3.2.1.1.94 PSA_KEY_TYPE_AES

Key for a cipher, AEAD or MAC algorithm based on the AES block cipher.

The size of the key is related to the AES algorithm variant. For algorithms except the XTS block cipher mode, the following key sizes are used:

- AES-128 uses a 16-byte key : key_bits = 128
- AES-192 uses a 24-byte key : key_bits = 192
- AES-256 uses a 32-byte key : key_bits = 256

For the XTS block cipher mode (PSA_ALG_XTS), the following key sizes are used:

- AES-128-XTS uses two 16-byte keys : key_bits = 256
- AES-192-XTS uses two 24-byte keys : key_bits = 384
- AES-256-XTS uses two 32-byte keys : key_bits = 512

The AES block cipher is defined in FIPS Publication 197: Advanced Encryption Standard (AES) [FIPS197].

3.2.1.1.95 PSA_KEY_TYPE_ARC4

Key for the ARC4 stream cipher.

Warning:
The ARC4 cipher is weak and deprecated and is only recommended for use in legacy protocols.

The ARC4 cipher supports key sizes between 40 and 2048 bits, that are multiples of 8. (5 to 256 bytes)

Use algorithm PSA_ALG_STREAM_CIPHER to use this key with the ARC4 cipher.

3.2.1.1.96 PSA_KEY_TYPE_ARIA

Key for a cipher, AEAD or MAC algorithm based on the ARIA block cipher.

The size of the key is related to the ARIA algorithm variant. For algorithms except the XTS block cipher mode, the following key sizes are used:

- ARIA-128 uses a 16-byte key : key_bits = 128
- ARIA-192 uses a 24-byte key : key_bits = 192
- ARIA-256 uses a 32-byte key : key_bits = 256

For the XTS block cipher mode (PSA_ALG_XTS), the following key sizes are used:

- ARIA-128-XTS uses two 16-byte keys : key_bits = 256
- ARIA-192-XTS uses two 24-byte keys : key_bits = 384
- ARIA-256-XTS uses two 32-byte keys : key_bits = 512

The ARIA block cipher is defined in A Description of the ARIA Encryption Algorithm [RFC5794].

Compatible algorithms:

- PSA_ALG_CBC_MAC
- PSA_ALG_CMAC
- PSA_ALG_CTR
- PSA_ALG_CFB
- PSA_ALG_OFB
- PSA_ALG_XTS
- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_ECB_NO_PADDING
- PSA_ALG_CCM
- PSA_ALG_GCM

3.2.1.1.97 PSA_KEY_TYPE_CAMELLIA

Key for a cipher, AEAD or MAC algorithm based on the Camellia block cipher.

The size of the key is related to the Camellia algorithm variant. For algorithms except the XTS block cipher mode, the following key sizes are used:

- Camellia-128 uses a 16-byte key : key_bits = 128
- Camellia-192 uses a 24-byte key : key_bits = 192
- Camellia-256 uses a 32-byte key : key_bits = 256

For the XTS block cipher mode (PSA_ALG_XTS), the following key sizes are used:

- Camellia-128-XTS uses two 16-byte keys : key_bits = 256

- Camellia-192-XTS uses two 24-byte keys : key_bits = 384
- Camellia-256-XTS uses two 32-byte keys : key_bits = 512

The Camellia block cipher is defined in Specification of Camellia — a 128-bit Block Cipher [NTT-CAM] and also described in A Description of the Camellia Encryption Algorithm [RFC3713].

3.2.1.1.98 PSA_KEY_TYPE_CHACHA20

Key for the ChaCha20 stream cipher or the ChaCha20-Poly1305 AEAD algorithm.

The ChaCha20 key size is 256 bits (32 bytes).

- Use algorithm PSA_ALG_STREAM_CIPHER to use this key with the ChaCha20 cipher for unauthenticated encryption. See PSA_ALG_STREAM_CIPHER for details of this algorithm.
- Use algorithm PSA_ALG_CHACHA20_POLY1305 to use this key with the ChaCha20 cipher and Poly1305 authenticator for AEAD. See PSA_ALG_CHACHA20_POLY1305 for details of this algorithm.

3.2.1.1.99 PSA_KEY_TYPE_DERIVE

A secret for key derivation.

The key policy determines which key derivation algorithm the key can be used for.

The bit size of a secret for key derivation must be a non-zero multiple of 8. The maximum size of a secret for key derivation is IMPLEMENTATION DEFINED.

3.2.1.1.100 PSA_KEY_TYPE_DES

Key for a cipher or MAC algorithm based on DES or 3DES (Triple-DES).

The size of the key determines which DES algorithm is used:

- Single DES uses an 8-byte key : key_bits = 64
- 2-key 3DES uses a 16-byte key : key_bits = 128
- 3-key 3DES uses a 24-byte key : key_bits = 192

Warning:

> Single DES and 2-key 3DES are weak and strongly deprecated and are only recommended for decrypting legacy data.
>
> 3-key 3DES is weak and deprecated and is only recommended for use in legacy protocols.

The DES and 3DES block ciphers are defined in NIST Special Publication 800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher [SP800-67].

3.2.1.1.101 macro PSA_KEY_TYPE_DH_GET_FAMILY

PSA_KEY_TYPE_DH_GET_FAMILY(type)

Extract the group family from a Diffie-Hellman key type.

Parameters

- **type** – A Diffie-Hellman key type (value of *typedef psa_key_type_t* such that PSA_KEY_TYPE_IS_DH(type) is true).

3.2.1.1.101.1 Return

typedef psa_dh_family_t

The Diffie-Hellman group family id, if `type` is a supported Diffie-Hellman key. Unspecified if `type` is not a supported Diffie-Hellman key.

3.2.1.1.102 macro PSA_KEY_TYPE_DH_KEY_PAIR

PSA_KEY_TYPE_DH_KEY_PAIR(group)

Finite-field Diffie-Hellman key pair: both the private key and public key.

Parameters

- **group** – A value of *typedef psa_dh_family_t* that identifies the Diffie-Hellman group family to be used.

3.2.1.1.103 macro PSA_KEY_TYPE_DH_PUBLIC_KEY

PSA_KEY_TYPE_DH_PUBLIC_KEY(group)

Finite-field Diffie-Hellman public key.

Parameters

- **group** – A value of *typedef psa_dh_family_t* that identifies the Diffie-Hellman group family to be used.

3.2.1.1.104 macro PSA_KEY_TYPE_ECC_GET_FAMILY

PSA_KEY_TYPE_ECC_GET_FAMILY(type)

Extract the curve family from an elliptic curve key type.

Parameters

- **type** – An elliptic curve key type (value of *typedef psa_key_type_t* such that PSA_KEY_TYPE_IS_ECC(type) is true).

3.2.1.1.104.1 Return

typedef psa_ecc_family_t

The elliptic curve family id, if `type` is a supported elliptic curve key. Unspecified if `type` is not a supported elliptic curve key.

3.2.1.1.105 macro PSA_KEY_TYPE_ECC_KEY_PAIR

PSA_KEY_TYPE_ECC_KEY_PAIR(curve)

Elliptic curve key pair: both the private and public key.

Parameters

- **curve** – A value of *typedef psa_ecc_family_t* that identifies the ECC curve family to be used.

3.2.1.1.106 macro PSA_KEY_TYPE_ECC_PUBLIC_KEY

PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve)

Elliptic curve public key.

Parameters

- **curve** – A value of *typedef psa_ecc_family_t* that identifies the ECC curve family to be used.

3.2.1.1.107 PSA_KEY_TYPE_HMAC

HMAC key.

The key policy determines which underlying hash algorithm the key can be used for.

The bit size of an HMAC key must be a non-zero multiple of 8. An HMAC key is typically the same size as the output of the underlying hash algorithm. An HMAC key that is longer than the block size of the underlying hash algorithm will be hashed before use.

When an HMAC key is created that is longer than the block size, it is implementation defined whether the implementation stores the original HMAC key, or the hash of the HMAC key. If the hash of the key is stored, the key size reported by *psa_get_key_attributes()* will be the size of the hashed key.

Note:

PSA_HASH_LENGTH(alg) provides the output size of hash algorithm alg, in bytes.

PSA_HASH_BLOCK_LENGTH(alg) provides the block size of hash algorithm alg, in bytes.

3.2.1.1.108 macro PSA_KEY_TYPE_IS_ASYMMETRIC

`PSA_KEY_TYPE_IS_ASYMMETRIC`(type)

Whether a key type is asymmetric: either a key pair or a public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.108.1 Description

See RSA keys for a list of asymmetric key types.

3.2.1.1.109 macro PSA_KEY_TYPE_IS_DH

`PSA_KEY_TYPE_IS_DH`(type)

Whether a key type is a Diffie-Hellman key, either a key pair or a public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.110 macro PSA_KEY_TYPE_IS_DH_KEY_PAIR

`PSA_KEY_TYPE_IS_DH_KEY_PAIR`(type)

Whether a key type is a Diffie-Hellman key pair.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.111 macro PSA_KEY_TYPE_IS_DH_PUBLIC_KEY

`PSA_KEY_TYPE_IS_DH_PUBLIC_KEY`(type)

Whether a key type is a Diffie-Hellman public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.112 macro PSA_KEY_TYPE_IS_ECC

`PSA_KEY_TYPE_IS_ECC`(type)

Whether a key type is an elliptic curve key, either a key pair or a public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.113 macro PSA_KEY_TYPE_IS_ECC_KEY_PAIR

PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type)

Whether a key type is an elliptic curve key pair.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.114 macro PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY

PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type)

Whether a key type is an elliptic curve public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.115 macro PSA_KEY_TYPE_IS_KEY_PAIR

PSA_KEY_TYPE_IS_KEY_PAIR(type)

Whether a key type is a key pair containing a private part and a public part.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.116 macro PSA_KEY_TYPE_IS_PUBLIC_KEY

PSA_KEY_TYPE_IS_PUBLIC_KEY(type)

Whether a key type is the public part of a key pair.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.117 macro PSA_KEY_TYPE_IS_RSA

PSA_KEY_TYPE_IS_RSA(type)

Whether a key type is an RSA key. This includes both key pairs and public keys.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.118 macro PSA_KEY_TYPE_IS_RSA_KEY_PAIR

PSA_KEY_TYPE_IS_RSA_KEY_PAIR(type)

Whether a key type is an RSA key pair.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.119 macro PSA_KEY_TYPE_IS_RSA_PUBLIC_KEY

PSA_KEY_TYPE_IS_RSA_PUBLIC_KEY(type)

Whether a key type is an RSA public key.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.120 macro PSA_KEY_TYPE_IS_UNSTRUCTURED

PSA_KEY_TYPE_IS_UNSTRUCTURED(type)

Whether a key type is an unstructured array of bytes.

Parameters

- **type** – A key type (value of *typedef psa_key_type_t*).

3.2.1.1.120.1 Description

This encompasses both symmetric keys and non-key data.

See Symmetric keys for a list of symmetric key types.

3.2.1.1.121 macro PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY

PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type)

The key pair type corresponding to a public key type.

Parameters

- **type** – A public key type or key pair type.

3.2.1.1.121.1 Description

If type is a key pair type, it will be left unchanged.

3.2.1.1.121.2 Return

The corresponding key pair type. If `type` is not a public key or a key pair, the return value is undefined.

3.2.1.1.122 PSA_KEY_TYPE_NONE

An invalid key type value.

Zero is not the encoding of any key type.

3.2.1.1.123 PSA_KEY_TYPE_PASSWORD

A low-entropy secret for password hashing or key derivation.

This key type is suitable for passwords and passphrases which are typically intended to be memorizable by humans, and have a low entropy relative to their size. It can be used for randomly generated or derived keys with maximum or near-maximum entropy, but PSA_KEY_TYPE_DERIVE is more suitable for such keys. It is not suitable for passwords with extremely low entropy, such as numerical PINs.

These keys can be used in the PSA_KEY_DERIVATION_INPUT_PASSWORD input step of key derivation algorithms. Algorithms that accept such an input were designed to accept low-entropy secret and are known as password hashing or key stretching algorithms.

These keys cannot be used in the PSA_KEY_DERIVATION_INPUT_SECRET input step of key derivation algorithms, as the algorithms expect such an input to have high entropy.

The key policy determines which key derivation algorithm the key can be used for, among the permissible subset defined above.

Compatible algorithms:

- *PSA_ALG_PBKDF2_HMAC()* (password input)
- PSA_ALG_PBKDF2_AES_CMAC_PRF_128 (password input)

3.2.1.1.124 PSA_KEY_TYPE_PASSWORD_HASH

A secret value that can be used to verify a password hash.

The key policy determines which key derivation algorithm the key can be used for, among the same permissible subset as for PSA_KEY_TYPE_PASSWORD.

Compatible algorithms:

- *PSA_ALG_PBKDF2_HMAC()* (key output and verification)
- PSA_ALG_PBKDF2_AES_CMAC_PRF_128 (key output and verification)

3.2.1.1.125 PSA_KEY_TYPE_PEPPER

A secret value that can be used when computing a password hash.

The key policy determines which key derivation algorithm the key can be used for, among the subset of algorithms that can use pepper.

Compatible algorithms:

- *PSA_ALG_PBKDF2_HMAC()* (salt input)
- PSA_ALG_PBKDF2_AES_CMAC_PRF_128 (salt input)

3.2.1.1.126 macro PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR

`PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type)`

The public key type corresponding to a key pair type.

Parameters

- `type` – A public key type or key pair type.

3.2.1.1.126.1 Description

If type is a public key type, it will be left unchanged.

3.2.1.1.126.2 Return

The corresponding public key type. If `type` is not a public key or a key pair, the return value is undefined.

3.2.1.1.127 PSA_KEY_TYPE_RAW_DATA

Raw data.

A "key" of this type cannot be used for any cryptographic operation. Applications can use this type to store arbitrary data in the keystore.

The bit size of a raw key must be a non-zero multiple of 8. The maximum size of a raw key is IMPLEMENTATION DEFINED.

3.2.1.1.128 PSA_KEY_TYPE_RSA_KEY_PAIR

RSA key pair: both the private and public key.

3.2.1.1.129 PSA_KEY_TYPE_RSA_PUBLIC_KEY

RSA public key.

3.2.1.1.130 PSA_KEY_TYPE_SM4

Key for a cipher, AEAD or MAC algorithm based on the SM4 block cipher.

For algorithms except the XTS block cipher mode, the SM4 key size is 128 bits (16 bytes).

For the XTS block cipher mode (PSA_ALG_XTS), the SM4 key size is 256 bits (two 16-byte keys).

The SM4 block cipher is defined in GB/T 32907-2016: Information security technology — SM4 block cipher algorithm [PRC-SM4] and also described in The SM4 Blockcipher Algorithm And Its Modes Of Operations [IETF-SM4].

3.2.1.1.131 PSA_KEY_USAGE_CACHE

Permission for the implementation to cache the key.

This flag allows the implementation to make additional copies of the key material that are not in storage and not for the purpose of an ongoing operation. Applications can use it as a hint to keep the key around for repeated access.

An application can request that cached key material is removed from memory by calling *psa_purge_key()*.

The presence of this usage flag when creating a key is a hint:

- An implementation is not required to cache keys that have this usage flag.
- An implementation must not report an error if it does not cache keys.

If this usage flag is not present, the implementation must ensure key material is removed from memory as soon as it is not required for an operation or for maintenance of a volatile key.

This flag must be preserved when reading back the attributes for all keys, regardless of key type or implementation behavior.

3.2.1.1.132 PSA_KEY_USAGE_COPY

Permission to copy the key.

This flag allows the use of *psa_copy_key()* to make a copy of the key with the same policy or a more restrictive policy.

For lifetimes for which the key is located in a secure element which enforce the non-exportability of keys, copying a key outside the secure element also requires the usage flag PSA_KEY_USAGE_EXPORT. Copying the key inside the secure element is permitted with just PSA_KEY_USAGE_COPY if the secure element supports it. For keys with the lifetime PSA_KEY_LIFETIME_VOLATILE or PSA_KEY_LIFETIME_PERSISTENT, the usage flag PSA_KEY_USAGE_COPY is sufficient to permit the copy.

3.2.1.1.133 PSA_KEY_USAGE_DECRYPT

Permission to decrypt a message with the key. This flag allows the key to be used for a symmetric decryption operation, for an AEAD decryption-and-verification operation, or for an asymmetric decryption operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- *psa_cipher_decrypt()*
- *psa_cipher_decrypt_setup()*
- *psa_aead_decrypt()*
- *psa_aead_decrypt_setup()*
- *psa_asymmetric_decrypt()*

For a key pair, this concerns the private key.

3.2.1.1.134 PSA_KEY_USAGE_DERIVE

Permission to derive other keys from this key.

This flag allows the key to be used for a key derivation operation or for a key agreement operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- *psa_key_derivation_input_key()*
- *psa_key_derivation_key_agreement()*
- *psa_raw_key_agreement()*

3.2.1.1.135 PSA_KEY_USAGE_ENCRYPT

Permission to encrypt a message with the key.

This flag allows the key to be used for a symmetric encryption operation, for an AEAD encryption-and-authentication operation, or for an asymmetric encryption operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- *psa_cipher_encrypt()*
- *psa_cipher_encrypt_setup()*
- *psa_aead_encrypt()*
- *psa_aead_encrypt_setup()*
- *psa_asymmetric_encrypt()*

For a key pair, this concerns the public key.

3.2.1.1.136 PSA_KEY_USAGE_EXPORT

Permission to export the key.

This flag allows the use of *psa_export_key()* to export a key from the cryptoprocessor. A public ey or the public part of a key pair can always be exported regardless of the value of this permission flag.

This flag can also be required to copy a key using *psa_copy_key()* outside of a secure element. See also PSA_KEY_USAGE_COPY.

If a key does not have export permission, implementations must not allow the key to be exported in plain form from the cryptoprocessor, whether through *psa_export_key()* or through a proprietary interface. The key might still be exportable in a wrapped form, i.e. in a form where it is encrypted by another key.

3.2.1.1.137 PSA_KEY_USAGE_SIGN_HASH

Permission to sign a message hash with the key.

This flag allows the key to be used to sign a message hash as part of an asymmetric signature operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used when calling *psa_sign_hash()*.

This flag automatically sets PSA_KEY_USAGE_SIGN_MESSAGE: if an application sets the flag PSA_KEY_USAGE_SIGN_HASH when creating a key, then the key always has the permissions conveyed by PSA_KEY_USAGE_SIGN_MESSAGE, and the flag PSA_KEY_USAGE_SIGN_MESSAGE will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the private key.

3.2.1.1.138 PSA_KEY_USAGE_SIGN_MESSAGE

Permission to sign a message with the key.

This flag allows the key to be used for a MAC calculation operation or for an asymmetric message signature operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- *psa_mac_compute()*
- *psa_mac_sign_setup()*
- *psa_sign_message()*

For a key pair, this concerns the private key.

3.2.1.1.139 PSA_KEY_USAGE_VERIFY_DERIVATION

Permission to verify the result of a key derivation, including password hashing.

This flag allows the key to be used in a key derivation operation, if otherwise permitted by the key's type and policy.

This flag must be present on keys used with *psa_key_derivation_verify_key()*.

If this flag is present on all keys used in calls to *psa_key_derivation_input_key()* for a key derivation operation, then it permits calling *psa_key_derivation_verify_bytes()* or *psa_key_derivation_verify_key()* at the end of the operation.

3.2.1.1.140 PSA_KEY_USAGE_VERIFY_HASH

Permission to verify a message hash with the key.

This flag allows the key to be used to verify a message hash as part of an asymmetric signature verification operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used when calling *psa_verify_hash()*.

This flag automatically sets PSA_KEY_USAGE_VERIFY_MESSAGE: if an application sets the flag PSA_KEY_USAGE_VERIFY_HASH when creating a key, then the key always has the permissions conveyed by PSA_KEY_USAGE_VERIFY_MESSAGE, and the flag PSA_KEY_USAGE_VERIFY_MESSAGE will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the public key.

3.2.1.1.141 PSA_KEY_USAGE_VERIFY_MESSAGE

Permission to verify a message signature with the key.

This flag allows the key to be used for a MAC verification operation or for an asymmetric message signature verification operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- *psa_mac_verify()*
- *psa_mac_verify_setup()*
- *psa_verify_message()*

For a key pair, this concerns the public key.

3.2.1.2 Sizes

3.2.1.2.1 Introduction

This file contains the definitions of macros that are useful to compute buffer sizes. The signatures and semantics of these macros are standardized, but the definitions are not, because they depend on the available algorithms and, in some cases, on permitted tolerances on buffer sizes.

3.2.1.2.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.1.2.3 macro PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE

PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length)

A sufficient output buffer size for *psa_aead_decrypt()*, for any of the supported key types and AEAD algorithms.

Parameters

- **ciphertext_length** – Size of the ciphertext in bytes.

3.2.1.2.3.1 Description

Warning: Not supported

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

See also *PSA_AEAD_DECRYPT_OUTPUT_SIZE()*.

3.2.1.2.4 macro PSA_AEAD_DECRYPT_OUTPUT_SIZE

PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length)

The maximum size of the output of *psa_aead_decrypt()*, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
- **ciphertext_length** – Size of the ciphertext in bytes.

3.2.1.2.4.1 Description

Warning: Not supported

If the size of the plaintext buffer is at least this large, it is guaranteed that *psa_aead_decrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the plaintext might be smaller.

See also *PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE()*.

3.2.1.2.4.2 Return

The AEAD plaintext size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.5 macro PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE

PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length)

A sufficient output buffer size for *psa_aead_encrypt()*, for any of the supported key types and AEAD algorithms.

Parameters

- **plaintext_length** – Size of the plaintext in bytes.

3.2.1.2.5.1 Description

Warning: Not supported

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

See also *PSA_AEAD_ENCRYPT_OUTPUT_SIZE()*.

3.2.1.2.6 macro PSA_AEAD_ENCRYPT_OUTPUT_SIZE

PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length)

The maximum size of the output of *psa_aead_encrypt()*, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
- **plaintext_length** – Size of the plaintext in bytes.

3.2.1.2.6.1 Description

Warning: Not supported

If the size of the ciphertext buffer is at least this large, it is guaranteed that *psa_aead_encrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the ciphertext might be smaller.

See also *PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE()*.

3.2.1.2.6.2 Return

The AEAD ciphertext size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.7 PSA_AEAD_FINISH_OUTPUT_MAX_SIZE

A sufficient ciphertext buffer size for *psa_aead_finish()*, for any of the supported key types and AEAD algorithms.

Warning: Not supported

See also *PSA_AEAD_FINISH_OUTPUT_SIZE()*.

3.2.1.2.8 macro PSA_AEAD_FINISH_OUTPUT_SIZE

PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg)

A sufficient ciphertext buffer size for *psa_aead_finish()*.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.2.8.1 Description

Warning: Not supported

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

See also PSA_AEAD_FINISH_OUTPUT_MAX_SIZE.

3.2.1.2.8.2 Return

A sufficient ciphertext buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.9 macro PSA_AEAD_NONCE_LENGTH

PSA_AEAD_NONCE_LENGTH(key_type, alg)

The default nonce size for an AEAD algorithm, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.2.9.1 Description

Warning: Not supported

This macro can be used to allocate a buffer of sufficient size to store the nonce output from *psa_aead_generate_nonce()*.

See also PSA_AEAD_NONCE_MAX_SIZE.

3.2.1.2.9.2 Return

The default nonce size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.10 PSA_AEAD_NONCE_MAX_SIZE

The maximum nonce size for all supported AEAD algorithms, in bytes.

Warning: Not supported

See also *PSA_AEAD_NONCE_LENGTH()*.

3.2.1.2.11 macro PSA_AEAD_TAG_LENGTH

PSA_AEAD_TAG_LENGTH(key_type, key_bits, alg)

The length of a tag for an AEAD algorithm, in bytes.

Parameters

- **key_type** – The type of the AEAD key.
- **key_bits** – The size of the AEAD key in bits.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.2.11.1 Description

Warning: Not supported

This macro can be used to allocate a buffer of sufficient size to store the tag output from *psa_aead_finish()*.

See also PSA_AEAD_TAG_MAX_SIZE.

3.2.1.2.11.2 Return

The tag length for the specified algorithm and key. If the AEAD algorithm does not have an identified tag that can be distinguished from the rest of the ciphertext, return 0. If the AEAD algorithm is not recognized, return 0. An implementation can return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support.

3.2.1.2.12 PSA_AEAD_TAG_MAX_SIZE

The maximum tag size for all supported AEAD algorithms, in bytes.

Warning: Not supported

See also *PSA_AEAD_TAG_LENGTH()*.

3.2.1.2.13 macro PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE

PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length)

A sufficient output buffer size for *psa_aead_update()*, for any of the supported key types and AEAD algorithms.

Parameters

- **input_length** – Size of the input in bytes.

3.2.1.2.13.1 Description

Warning: Not supported

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

See also *PSA_AEAD_UPDATE_OUTPUT_SIZE()*.

3.2.1.2.14 macro PSA_AEAD_UPDATE_OUTPUT_SIZE

PSA_AEAD_UPDATE_OUTPUT_SIZE(key_type, alg, input_length)

A sufficient output buffer size for *psa_aead_update()*.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm alg.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
- **input_length** – Size of the input in bytes.

3.2.1.2.14.1 Description

Warning: Not supported

If the size of the output buffer is at least this large, it is guaranteed that *psa_aead_update()* 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_AEAD_UPDATE_OUTPUT_MAX_SIZE.

3.2.1.2.14.2 Return

A sufficient output buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.15 PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE

A sufficient plaintext buffer size for *psa_aead_verify()*, for any of the supported key types and AEAD algorithms.

Warning: Not supported

See also *PSA_AEAD_VERIFY_OUTPUT_SIZE()*.

3.2.1.2.16 macro PSA_AEAD_VERIFY_OUTPUT_SIZE

PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg)

A sufficient plaintext buffer size for *psa_aead_verify()*.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm alg.
- **alg** – An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.2.16.1 Description

Warning: Not supported

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

See also PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE.

3.2.1.2.16.2 Return

A sufficient plaintext buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

3.2.1.2.17 PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE

A sufficient output buffer size for *psa_asymmetric_decrypt()*, for any supported asymmetric decryption.

Warning: Not supported

See also *PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE()*.

3.2.1.2.18 macro PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE

PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg)

Sufficient output buffer size for *psa_asymmetric_decrypt()*.

Parameters

- **key_type** – An asymmetric key type, either a key pair or a public key.
- **key_bits** – The size of the key in bits.
- **alg** – The asymmetric encryption algorithm.

3.2.1.2.18.1 Description

Warning: Not supported

This macro returns a sufficient buffer size for a plaintext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the plaintext might be smaller, depending on the algorithm.

Warning:

This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE.

3.2.1.2.18.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_decrypt()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

3.2.1.2.19 PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE

A sufficient output buffer size for *psa_asymmetric_encrypt()*, for any supported asymmetric encryption.

Warning: Not supported

See also *PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE()*.

3.2.1.2.20 macro PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE

PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg)

Sufficient output buffer size for *psa_asymmetric_encrypt()*.

Parameters

- **key_type** – An asymmetric key type, either a key pair or a public key.
- **key_bits** – The size of the key in bits.
- **alg** – The asymmetric encryption algorithm.

3.2.1.2.20.1 Description

Warning: Not supported

This macro returns a sufficient buffer size for a ciphertext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the ciphertext might be smaller, depending on the algorithm.

Warning:
This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE.

3.2.1.2.20.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_encrypt()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

3.2.1.2.21 PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE

The maximum size of a block cipher supported by the implementation.

See also *PSA_BLOCK_CIPHER_BLOCK_LENGTH()*.

3.2.1.2.22 macro PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE

PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length)

A sufficient output buffer size for *psa_cipher_decrypt()*, for any of the supported key types and cipher algorithms.

Parameters

- **input_length** – Size of the input in bytes.

3.2.1.2.22.1 Description

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

See also *PSA_CIPHER_DECRYPT_OUTPUT_SIZE()*.

3.2.1.2.23 macro PSA_CIPHER_DECRYPT_OUTPUT_SIZE

PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length)

The maximum size of the output of *psa_cipher_decrypt()*, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
- **input_length** – Size of the input in bytes.

3.2.1.2.23.1 Description

If the size of the output buffer is at least this large, it is guaranteed that *psa_cipher_decrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the output might be smaller.

See also PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE.

3.2.1.2.23.2 Return

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

3.2.1.2.24 macro PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE

PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length)

A sufficient output buffer size for *psa_cipher_encrypt()*, for any of the supported key types and cipher algorithms.

Parameters

- **input_length** – Size of the input in bytes.

3.2.1.2.24.1 Description

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

See also *PSA_CIPHER_ENCRYPT_OUTPUT_SIZE()*.

3.2.1.2.25 macro PSA_CIPHER_ENCRYPT_OUTPUT_SIZE

PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length)

The maximum size of the output of *psa_cipher_encrypt()*, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
- **input_length** – Size of the input in bytes.

3.2.1.2.25.1 Description

If the size of the output buffer is at least this large, it is guaranteed that *psa_cipher_encrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the output might be smaller.

See also PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE.

3.2.1.2.25.2 Return

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

3.2.1.2.26 PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE

A sufficient ciphertext buffer size for *psa_cipher_finish()*, for any of the supported key types and cipher algorithms.

Warning: Not supported

See also *PSA_CIPHER_FINISH_OUTPUT_SIZE()*.

3.2.1.2.27 macro PSA_CIPHER_FINISH_OUTPUT_SIZE

PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg)

A sufficient ciphertext buffer size for *psa_cipher_finish()*.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

3.2.1.2.27.1 Description

Warning: Not supported

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

See also PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE.

3.2.1.2.27.2 Return

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

3.2.1.2.28 macro PSA_CIPHER_IV_LENGTH

PSA_CIPHER_IV_LENGTH(key_type, alg)

The default IV size for a cipher algorithm, in bytes.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

3.2.1.2.28.1 Description

The IV that is generated as part of a call to *psa_cipher_encrypt()* is always the default IV length for the algorithm.

This macro can be used to allocate a buffer of sufficient size to store the IV output from *psa_cipher_generate_iv()* when using a multi-part cipher operation.

See also PSA_CIPHER_IV_MAX_SIZE.

3.2.1.2.28.2 Return

The default IV size for the specified key type and algorithm. If the algorithm does not use an IV, return 0. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

3.2.1.2.29 PSA_CIPHER_IV_MAX_SIZE

The maximum IV size for all supported cipher algorithms, in bytes.

See also *PSA_CIPHER_IV_LENGTH()*.

3.2.1.2.30 macro PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE

PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length)

A sufficient output buffer size for *psa_cipher_update()*, for any of the supported key types and cipher algorithms.

Parameters

- **input_length** – Size of the input in bytes.

3.2.1.2.30.1 Description

Warning: Not supported

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

See also *PSA_CIPHER_UPDATE_OUTPUT_SIZE()*.

3.2.1.2.31 macro PSA_CIPHER_UPDATE_OUTPUT_SIZE

PSA_CIPHER_UPDATE_OUTPUT_SIZE(key_type, alg, input_length)

A sufficient output buffer size for *psa_cipher_update()*.

Parameters

- **key_type** – A symmetric key type that is compatible with algorithm `alg`.
- **alg** – A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
- **input_length** – Size of the input in bytes.

3.2.1.2.31.1 Description

Warning: Not supported

If the size of the output buffer is at least this large, it is guaranteed that *psa_cipher_update()* 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_CIPHER_UPDATE_OUTPUT_MAX_SIZE.

3.2.1.2.31.2 Return

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

3.2.1.2.32 macro PSA_EXPORT_KEY_OUTPUT_SIZE

PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits)

Sufficient output buffer size for *psa_export_key()*.

Parameters

- **key_type** – A supported key type.
- **key_bits** – The size of the key in bits.

3.2.1.2.32.1 Description

The following code illustrates how to allocate enough memory to export a key by querying the key type and size at runtime.

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
psa_status_t status;
status = psa_get_key_attributes(key, &attributes);
if (status != PSA_SUCCESS)
    handle_error(...);
psa_key_type_t key_type = psa_get_key_type(&attributes);
size_t key_bits = psa_get_key_bits(&attributes);
size_t buffer_size = PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits);
psa_reset_key_attributes(&attributes);
uint8_t *buffer = malloc(buffer_size);
if (buffer == NULL)
    handle_error(...);
size_t buffer_length;
status = psa_export_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS)
    handle_error(...);
```

See also PSA_EXPORT_KEY_PAIR_MAX_SIZE and PSA_EXPORT_PUBLIC_KEY_MAX_SIZE.

3.2.1.2.32.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_export_key()* or *psa_export_public_key()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

3.2.1.2.33 PSA_EXPORT_KEY_PAIR_MAX_SIZE

Sufficient buffer size for exporting any asymmetric key pair.

This value must be a sufficient buffer size when calling *psa_export_key()* to export any asymmetric key pair that is supported by the implementation, regardless of the exact key type and key size.

See also *PSA_EXPORT_KEY_OUTPUT_SIZE()*.

3.2.1.2.34 PSA_EXPORT_PUBLIC_KEY_MAX_SIZE

Sufficient buffer size for exporting any asymmetric public key.

This value must be a sufficient buffer size when calling *psa_export_key()* or *psa_export_public_key()* to export any asymmetric public key that is supported by the implementation, regardless of the exact key type and key size.

See also *PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE()*.

3.2.1.2.35 macro PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE

PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(key_type, key_bits)

Sufficient output buffer size for *psa_export_public_key()*.

Parameters

- **key_type** – A public key or key pair key type.
- **key_bits** – The size of the key in bits.

3.2.1.2.35.1 Description

The following code illustrates how to allocate enough memory to export a public key by querying the key type and size at runtime.

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
psa_status_t status;
status = psa_get_key_attributes(key, &attributes);
if (status != PSA_SUCCESS)
    handle_error(...);
psa_key_type_t key_type = psa_get_key_type(&attributes);
size_t key_bits = psa_get_key_bits(&attributes);
```

(continues on next page)

(continued from previous page)

```
size_t buffer_size = PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(key_type, key_bits);
psa_reset_key_attributes(&attributes);
uint8_t *buffer = malloc(buffer_size);
if (buffer == NULL)
    handle_error(...);
size_t buffer_length;
status = psa_export_public_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS)
    handle_error(...);
```

See also PSA_EXPORT_PUBLIC_KEY_MAX_SIZE.

3.2.1.2.35.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_export_public_key()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

If the parameters are valid and supported, it is recommended that this macro returns the same result as PSA_EXPORT_KEY_OUTPUT_SIZE(PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(key_type), key_bits).

3.2.1.2.36 macro PSA_HASH_BLOCK_LENGTH

PSA_HASH_BLOCK_LENGTH(alg)

The input block size of a hash algorithm, in bytes.

Parameters

- **alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

3.2.1.2.36.1 Description

Hash algorithms process their input data in blocks. Hash operations will retain any partial blocks until they have enough input to fill the block or until the operation is finished.

This affects the output from *psa_hash_suspend()*.

3.2.1.2.36.2 Return

The block size in bytes for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation can return either 0 or the correct size for a hash algorithm that it recognizes, but does not support.

3.2.1.2.37 macro PSA_HASH_LENGTH

PSA_HASH_LENGTH(alg)

The size of the output of *psa_hash_compute()* and *psa_hash_finish()*, in bytes.

Parameters

- **alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true), or an HMAC algorithm (PSA_ALG_HMAC(hash_alg) where hash_alg is a hash algorithm).

3.2.1.2.37.1 Description

This is also the hash length that *psa_hash_compare()* and *psa_hash_verify()* expect.

See also PSA_HASH_MAX_SIZE.

3.2.1.2.37.2 Return

The hash length for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation can return either 0 or the correct size for a hash algorithm that it recognizes, but does not support.

3.2.1.2.38 PSA_HASH_MAX_SIZE

Maximum size of a hash.

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of a hash supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also *PSA_HASH_LENGTH()*.

3.2.1.2.39 PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH

The size of the algorithm field that is part of the output of *psa_hash_suspend()*, in bytes.

Warning: Not supported

Applications can use this value to unpack the hash suspend state that is output by *psa_hash_suspend()*.

3.2.1.2.40 macro PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH

PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg)

The size of the hash-state field that is part of the output of *psa_hash_suspend()*, in bytes.

Parameters

- **alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

3.2.1.2.40.1 Description

Applications can use this value to unpack the hash suspend state that is output by *psa_hash_suspend()*.

3.2.1.2.40.2 Return

The size, in bytes, of the hash-state field of the hash suspend state for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation can return either 0 or the correct size for a hash algorithm that it recognizes, but does not support.

3.2.1.2.41 macro PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH

PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg)

The size of the input-length field that is part of the output of *psa_hash_suspend()*, in bytes.

Parameters

- **alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

3.2.1.2.41.1 Description

Applications can use this value to unpack the hash suspend state that is output by *psa_hash_suspend()*.

3.2.1.2.41.2 Return

The size, in bytes, of the input-length field of the hash suspend state for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation can return either 0 or the correct size for a hash algorithm that it recognizes, but does not support.

3.2.1.2.42 PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE

A sufficient hash suspend state buffer size for *psa_hash_suspend()*, for any supported hash algorithms.

Warning: Not supported

See also *PSA_HASH_SUSPEND_OUTPUT_SIZE()*.

3.2.1.2.43 macro PSA_HASH_SUSPEND_OUTPUT_SIZE

PSA_HASH_SUSPEND_OUTPUT_SIZE(alg)

A sufficient hash suspend state buffer size for *psa_hash_suspend()*.

Parameters

- **alg** – A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

3.2.1.2.43.1 Description

If the size of the hash state buffer is at least this large, it is guaranteed that *psa_hash_suspend()* 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_HASH_SUSPEND_OUTPUT_MAX_SIZE.

3.2.1.2.43.2 Return

A sufficient output size for the algorithm. If the hash algorithm is not recognized, or is not supported by *psa_hash_suspend()*, return 0. An implementation can return either 0 or a correct size for a hash algorithm that it recognizes, but does not support.

For a supported hash algorithm alg, the following expression is true:

```
PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) == PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH +
                                     PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_
↪LENGTH(alg) +
                                     PSA_HASH_SUSPEND_HASH_STATE_FIELD_
↪LENGTH(alg) +
                                     PSA_HASH_BLOCK_LENGTH(alg) - 1
```

3.2.1.2.44 macro PSA_MAC_TRUNCATED_LENGTH

PSA_MAC_TRUNCATED_LENGTH(alg)

Size of the truncated MAC algorithm in bytes.

Parameters

- **alg** – A MAC algorithm (such that PSA_ALG_IS_MAC_TRUNCATED(alg) is true).

3.2.1.2.44.1 Return

The MAC truncated length for the specified algorithm. 0 if the algorithm is not a MAC or a truncated MAC algorithm.

3.2.1.2.45 macro PSA_HMAC_LENGTH

PSA_HMAC_LENGTH(alg)

Size of the HMAC output length in bytes.

Parameters

- **alg** – A MAC algorithm (such that PSA_ALG_IS_HMAC(alg) is true).

3.2.1.2.45.1 Return

The MAC length for the specified algorithm. 0 if the MAC algorithm is not HMAC.

3.2.1.2.46 macro PSA_MAC_LENGTH

PSA_MAC_LENGTH(key_type, key_bits, alg)

The size of the output of *psa_mac_compute()* and *psa_mac_sign_finish()*, in bytes.

Parameters

- **key_type** – The type of the MAC key.
- **key_bits** – The size of the MAC key in bits.
- **alg** – A MAC algorithm (such that PSA_ALG_IS_MAC(alg) is true).

3.2.1.2.46.1 Description

This is also the MAC length that *psa_mac_verify()* and *psa_mac_verify_finish()* expect.

See also PSA_MAC_MAX_SIZE.

3.2.1.2.46.2 Return

The MAC length for the specified algorithm with the specified key parameters.

0 if the MAC algorithm is not recognized.

Either 0 or the correct length for a MAC algorithm that the implementation recognizes, but does not support.

3.2.1.2.47 PSA_MAC_MAX_SIZE

Maximum size of a MAC.

This macro must expand to a compile-time constant integer. The maximum MAC size is based on the maximum hash size supported by HMAC

See also *PSA_MAC_LENGTH()*.

3.2.1.2.48 PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE

Maximum size of the output from *psa_raw_key_agreement()*.

Warning: Not supported

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of the output any raw key agreement algorithm supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also *PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE()*.

3.2.1.2.49 macro PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE

PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(key_type, key_bits)

Sufficient output buffer size for *psa_raw_key_agreement()*.

Parameters

- **key_type** – A supported key type.
- **key_bits** – The size of the key in bits.

3.2.1.2.49.1 Description

Warning: Not supported

This macro returns a compile-time constant if its arguments are compile-time constants.

Warning:

This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE.

3.2.1.2.49.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_raw_key_agreement()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

3.2.1.2.50 PSA_ECC_SIGNATURE_SIZE

Size of an elliptic curve signature.

`key_bits`: The size of the key in bits.

3.2.1.2.51 PSA_SIGNATURE_MAX_SIZE

Maximum size of an asymmetric signature.

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of an asymmetric signature supported by the implementation, in bytes. The value must not be smaller than this maximum.

3.2.1.2.52 macro PSA_SIGN_OUTPUT_SIZE

`PSA_SIGN_OUTPUT_SIZE`(key_type, key_bits, alg)

Sufficient signature buffer size for *psa_sign_message()* and *psa_sign_hash()*.

Parameters

- **`key_type`** – An asymmetric key type. This can be a key pair type or a public key type.
- **`key_bits`** – The size of the key in bits.
- **`alg`** – The signature algorithm.

3.2.1.2.52.1 Description

This macro returns a sufficient buffer size for a signature using a key of the specified type and size, with the specified algorithm. Note that the actual size of the signature might be smaller, as some algorithms produce a variable-size signature.

See also PSA_SIGNATURE_MAX_SIZE.

3.2.1.2.52.2 Return

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_sign_message()* and *psa_sign_hash()* will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

3.2.1.2.53 PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE

This macro returns the maximum supported length of the PSK for the TLS-1.2 PSK-to-MS key derivation.

Warning: Not supported

This implementation-defined value specifies the maximum length for the PSK input used with a *PSA_ALG_TLS12_PSK_TO_MS()* key agreement algorithm.

Quoting Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) [RFC4279] §5.3:

> TLS implementations supporting these cipher suites MUST support arbitrary PSK identities up to 128 octets in length, and arbitrary PSKs up to 64 octets in length. Supporting longer identities and keys is RECOMMENDED.

Therefore, it is recommended that implementations define PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE with a value greater than or equal to 64.

3.2.1.3 Types

3.2.1.3.1 Introduction

This file declares types that encode errors, algorithms, key types, policies, etc.

3.2.1.3.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.1.3.3 typedef psa_algorithm_t

type `psa_algorithm_t`
Encoding of a cryptographic algorithm.

3.2.1.3.3.1 Description

This is a structured bitfield that identifies the category and type of algorithm. The range of algorithm identifier values is divided as follows:

- **0x00000000**
  Reserved as an invalid algorithm identifier.
- **0x00000001 - 0x7fffffff**
  Specification-defined algorithm identifiers. Algorithm identifiers defined by this standard always have bit 31 clear. Unallocated algorithm identifier values in this range are reserved for future use.
- **0x80000000 - 0xffffffff**
  Implementation-defined algorithm identifiers. Implementations that define additional algorithms must use an encoding with bit 31 set. The related support macros will be easier to write if these algorithm identifier encodings also respect the bitwise structure used by standard encodings.

For algorithms that can be applied to multiple key types, this identifier does not encode the key type. For example, for symmetric ciphers based on a block cipher, *`typedef psa_algorithm_t`* encodes the block cipher mode and the padding mode while the block cipher itself is encoded via *`typedef psa_key_type_t`*.

3.2.1.3.3.2 Values

- PSA_ALG_ANY_HASH
- PSA_ALG_CBC_MAC
- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_CCM
- PSA_ALG_CFB
- PSA_ALG_CHACHA20_POLY1305
- PSA_ALG_CMAC
- PSA_ALG_CTR
- PSA_ALG_ECB_NO_PADDING
- PSA_ALG_ECDH
- PSA_ALG_ECDSA_ANY
- PSA_ALG_FFDH
- PSA_ALG_GCM
- PSA_ALG_MD2
- PSA_ALG_MD4
- PSA_ALG_MD5

- PSA_ALG_NONE
- PSA_ALG_OFB
- PSA_ALG_RIPEMD160
- PSA_ALG_RSA_PKCS1V15_CRYPT
- PSA_ALG_RSA_PKCS1V15_SIGN_RAW
- PSA_ALG_SHA3_224
- PSA_ALG_SHA3_256
- PSA_ALG_SHA3_384
- PSA_ALG_SHA3_512
- PSA_ALG_SHA_1
- PSA_ALG_SHA_224
- PSA_ALG_SHA_256
- PSA_ALG_SHA_384
- PSA_ALG_SHA_512
- PSA_ALG_SHA_512_224
- PSA_ALG_SHA_512_256
- PSA_ALG_SM3
- PSA_ALG_STREAM_CIPHER
- PSA_ALG_XTS

3.2.1.3.4 typedef psa_dh_family_t

type **`psa_dh_family_t`**

The type of PSA finite-field Diffie-Hellman group family identifiers.

3.2.1.3.4.1 Description

The group family identifier is required to create an Diffie-Hellman key using the *`PSA_KEY_TYPE_DH_KEY_PAIR()`* or *`PSA_KEY_TYPE_DH_PUBLIC_KEY()`* macros.

The specific Diffie-Hellman group within a family is identified by the key_bits attribute of the key.

The range of Diffie-Hellman group family identifier values is divided as follows:

- **0x00 - 0x7f**
  DH group family identifiers defined by this standard. Unallocated values in this range are reserved for future use.
- **0x80 - 0xff**
  Implementations that define additional families must use an encoding in this range.

3.2.1.3.5 typedef psa_ecc_family_t

type **psa_ecc_family_t**

The type of PSA elliptic curve family identifiers.

3.2.1.3.5.1 Description

The curve identifier is required to create an ECC key using the *PSA_KEY_TYPE_ECC_KEY_PAIR()* or *PSA_KEY_TYPE_ECC_PUBLIC_KEY()* macros.

The specific ECC curve within a family is identified by the key_bits attribute of the key.

The range of Elliptic curve family identifier values is divided as follows:

- **0x00 - 0x7f**
  ECC family identifiers defined by this standard. Unallocated values in this range are reserved for future use.
- **0x80 - 0xff**
  Implementations that define additional families must use an encoding in this range.

3.2.1.3.6 typedef psa_key_derivation_step_t

type **psa_key_derivation_step_t**

Encoding of the step of a key derivation.

3.2.1.3.7 typedef psa_key_id_t

type **psa_key_id_t**

Key identifier.

3.2.1.3.7.1 Description

A key identifiers can be a permanent name for a persistent key, or a transient reference to volatile key.

3.2.1.3.8 typedef psa_key_lifetime_t

type **psa_key_lifetime_t**

Encoding of key lifetimes.

3.2.1.3.8.1 Description

The lifetime of a key indicates where it is stored and which application and system actions will create and destroy it.

Lifetime values have the following structure:

- Bits[7:0]: Persistence level

  This value indicates what device management actions can cause it to be destroyed. In particular, it indicates whether the key is volatile or persistent. See *typedef psa_key_persistence_t* for more information.

  PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) returns the persistence level for a key lifetime value.

- Bits[31:8]: Location indicator

  This value indicates where the key material is stored (or at least where it is accessible in cleartext) and where operations on the key are performed. See *typedef psa_key_location_t* for more information.

  PSA_KEY_LIFETIME_GET_LOCATION(lifetime) returns the location indicator for a key lifetime value.

Volatile keys (PSA_KEY_LIFETIME_VOLATILE) are automatically destroyed when the application instance terminates or on a power reset of the device. Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs, for example, a factory reset.

Persistent keys (PSA_KEY_LIFETIME_PERSISTENT) have a unique key identifier of type *typedef psa_key_id_t* per application instantiating the library. This identifier remains valid throughout the lifetime of the key, even if the application instance that created the key terminates.

3.2.1.3.9 typedef psa_key_location_t

type **psa_key_location_t**

Encoding of key location indicators.

3.2.1.3.9.1 Description

If an implementation of this API can make calls to external cryptoprocessors such as secure elements, the location of a key indicates which secure element performs the operations on the key. If the key material is not stored persistently inside the secure element, it must be stored in a wrapped form such that only the secure element can access the key material in cleartext.

Values for location indicators defined by this specification are shown below.

Location indicator	Definition
0	Primary local storage. The primary local storage is typically the same storage area that contains the key metadata.
1	Primary secure element. HSM or ELE Secure Subsystems are primary secure elements. As a guideline, secure elements may provide higher resistance against side channel and physical attacks than the primary local storage, but may have restrictions on supported key types, sizes, policies and operations and may have different performance characteristics.
2 - 0x7fffff	Other locations defined by a PSA specification. The PSA Cryptography API does not currently assign any meaning to these locations, but future versions of this specification or other PSA specifications may do so.
0x800000 - 0xffffff	Vendor-defined locations. No PSA specification will assign a meaning to locations in this range.

3.2.1.3.9.2 Note

Key location indicators are 24-bit values. Key management interfaces operate on lifetimes (see *typedef psa_key_lifetime_t*), and encode the location as the upper 24 bits of a 32-bit value.

3.2.1.3.10 typedef psa_key_persistence_t

type **psa_key_persistence_t**

Encoding of key persistence levels.

3.2.1.3.10.1 Description

What distinguishes different persistence levels is which device management events can cause keys to be destroyed. For example, power reset, transfer of device ownership, or a factory reset are device management events that can affect keys at different persistence levels. The specific management events which affect persistent keys at different levels is outside the scope of the PSA Cryptography specification.

Values for persistence levels defined by this specification are shown below.

Persistence level	Definition
0 = PSA_KEY_PERSISTENCE_VOLATILI	Volatile key. A volatile key is automatically destroyed by the implementation when the application instance terminates. In particular, a volatile key is automatically destroyed on a power reset of the device.
1 = PSA_KEY_PERSISTENCE_DEFAULT	Persistent key with a default lifetime. Applications should use this value if they have no specific needs that are only met by implementation-specific features.
2 - 127	Persistent key with a PSA-specified lifetime. The PSA Cryptography specification does not define the meaning of these values, but other PSA specifications may do so.
128 - 254	Persistent key with a vendor-specified lifetime. No PSA specification will define the meaning of these values, so implementations may choose the meaning freely. As a guideline, higher persistence levels should cause a key to survive more management events than lower levels.
255 = PSA_KEY_PERSISTENCE_READ_(	Read-only or write-once key. A key with this persistence level cannot be destroyed. Note that keys that are read-only due to policy restrictions rather than due to physical limitations should not have this persistence level.

3.2.1.3.10.2 Note

Key persistence levels are 8-bit values. Key management interfaces operate on lifetimes (see *typedef psa_key_lifetime_t*), and encode the persistence value as the lower 8 bits of a 32-bit value.

3.2.1.3.11 typedef psa_key_type_t

type **psa_key_type_t**

Encoding of a key type.

3.2.1.3.11.1 Description

This is a structured bitfield that identifies the category and type of key. The range of key type values is divided as follows:

- **PSA_KEY_TYPE_NONE == 0**
  Reserved as an invalid key type.
- **0x0001 - 0x7fff**
  Specification-defined key types. Key types defined by this standard always have bit 15 clear. Unallocated key type values in this range are reserved for future use.
- **0x8000 - 0xffff**
  No additional key type is defined.

3.2.1.3.12 typedef psa_key_usage_t

type **psa_key_usage_t**

Encoding of permitted usage on a key.

3.2.1.4 Structures

3.2.1.4.1 Introduction

This file contains the definitions of the data structures exposed by the PSA Cryptography API.

3.2.1.4.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.1.4.3 PSA_AEAD_OPERATION_INIT

This macro returns a suitable initializer for an AEAD operation object of type *typedef psa_aead_operation_t*.

3.2.1.4.4 PSA_CIPHER_OPERATION_INIT

This macro returns a suitable initializer for a cipher operation object of type *typedef psa_cipher_operation_t*.

3.2.1.4.5 PSA_HASH_OPERATION_INIT

This macro returns a suitable initializer for a hash operation object of type *typedef psa_hash_operation_t*.

3.2.1.4.6 PSA_KEY_DERIVATION_OPERATION_INIT

This macro returns a suitable initializer for a key derivation operation object of type *typedef psa_key_derivation_operation_t*.

3.2.1.4.7 PSA_MAC_OPERATION_INIT

This macro returns a suitable initializer for a MAC operation object of type *typedef psa_mac_operation_t*.

3.2.1.5 Functions

3.2.1.5.1 Introduction

This file contains the delarations of the crypto functions supported by the PSA Cryptography API.

3.2.1.5.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.1.5.3 typedef psa_aead_operation_t

type **psa_aead_operation_t**
The type of the state object for multi-part AEAD operations.

3.2.1.5.3.1 Description

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

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

```
psa_aead_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_aead_operation_t operation;
```

- Initialize the object to the initializer PSA_AEAD_OPERATION_INIT, for example:

```
psa_aead_operation_t operation = PSA_AEAD_OPERATION_INIT;
```

- Assign the result of the function *psa_aead_operation_init()* to the object, for example:

```
psa_aead_operation_t operation;
```

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

3.2.1.5.4 typedef psa_cipher_operation_t

type **psa_cipher_operation_t**

The type of the state object for multi-part cipher operations.

3.2.1.5.4.1 Description

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

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

```
psa_cipher_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_cipher_operation_t operation;
```

- Initialize the object to the initializer PSA_CIPHER_OPERATION_INIT, for example:

```
psa_cipher_operation_t operation = PSA_CIPHER_OPERATION_INIT;
```

- Assign the result of the function *psa_cipher_operation_init()* to the object, for example:

```
psa_cipher_operation_t operation;
operation = psa_cipher_operation_init();
```

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

3.2.1.5.5 typedef psa_hash_operation_t

type **psa_hash_operation_t**

The type of the state object for multi-part hash operations.

3.2.1.5.5.1 Description

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

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

```
psa_hash_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_hash_operation_t operation;
```

- Initialize the object to the initializer PSA_HASH_OPERATION_INIT, for example:

```
psa_hash_operation_t operation = PSA_HASH_OPERATION_INIT;
```

- Assign the result of the function *psa_hash_operation_init()* to the object, for example:

```
psa_hash_operation_t operation;
operation = psa_hash_operation_init();
```

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

3.2.1.5.6 typedef psa_key_attributes_t

type **psa_key_attributes_t**

The type of an object containing key attributes.

3.2.1.5.6.1 Description

This is the object that represents the metadata of a key object. Metadata that can be stored in attributes includes:

- The location of the key in storage, indicated by its key identifier and its lifetime.
- The key's policy, comprising usage flags and a specification of the permitted algorithm(s).
- Information about the key itself: the key type and its size.
- Implementation specific attributes.

The actual key material is not considered an attribute of a key. Key attributes do not contain information that is generally considered highly confidential.

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

Each attribute of this object is set with a function `psa_set_key_xxx()` and retrieved with a function `psa_get_key_xxx()`.

An attribute object can contain references to auxiliary resources, for example pointers to allocated memory or indirect references to pre-calculated values. In order to free such resources, the application must call *psa_reset_key_attributes()*. As an exception, calling *psa_reset_key_attributes()* on an attribute object is optional if the object has only been modified by the following functions since it was initialized or last reset with *psa_reset_key_attributes()*:

- *psa_set_key_id()*
- *psa_set_key_lifetime()*
- *psa_set_key_type()*
- *psa_set_key_bits()*
- *psa_set_key_usage_flags()*
- *psa_set_key_algorithm()*

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

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

```
psa_key_attributes_t attributes;
memset(&attributes, 0, sizeof(attributes));
```

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

```
static psa_key_attributes_t attributes;
```

- Initialize the object to the initializer PSA_KEY_ATTRIBUTES_INIT, for example:

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
```

- Assign the result of the function *psa_key_attributes_init()* to the object, for example:

```
psa_key_attributes_t attributes;
attributes = psa_key_attributes_init();
```

A freshly initialized attribute object contains the following values:

Attribute	Value
lifetime	PSA_KEY_LIFETIME_VOLATILE.
key identifier	PSA_KEY_ID_NULL - which is not a valid key identifier.
type	PSA_KEY_TYPE_NONE - meaning that the type is unspecified.
key size	0 - meaning that the size is unspecified.
usage flags	0 - which allows no usage except exporting a public key.
algorithm	PSA_ALG_NONE - which does not allow cryptographic usage, but allows exporting.

Usage:

A typical sequence to create a key is as follows:

1. Create and initialize an attribute object.
2. If the key is persistent, call *psa_set_key_id()*. Also call *psa_set_key_lifetime()* to place the key in a non-default location.
3. Set the key policy with *psa_set_key_usage_flags()* and *psa_set_key_algorithm()*.
4. Set the key type with *psa_set_key_type()*. Skip this step if copying an existing key with *psa_copy_key()*.
5. When generating a random key with *psa_generate_key()* or deriving a key with *psa_key_derivation_output_key()*, set the desired key size with *psa_set_key_bits()*.
6. Call a key creation function: *psa_import_key()*, *psa_generate_key()*, *psa_key_derivation_output_key()* or *psa_copy_key()*. This function reads the attribute object, creates a key with these attributes, and outputs an identifier for the newly created key.
7. Optionally call *psa_reset_key_attributes()*, now that the attribute object is no longer needed. Currently this call is not required as the attributes defined in this specification do not require additional resources beyond the object itself.

A typical sequence to query a key's attributes is as follows:

1. Call *psa_get_key_attributes()*.
2. Call `psa_get_key_xxx()` functions to retrieve the required attribute(s).
3. Call *psa_reset_key_attributes()* to free any resources that can be used by the attribute object.

Once a key has been created, it is impossible to change its attributes.

3.2.1.5.7 typedef psa_key_derivation_operation_t

type **psa_key_derivation_operation_t**

The type of the state object for key derivation operations.

3.2.1.5.7.1 Description

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

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

```
psa_key_derivation_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_key_derivation_operation_t operation;
```

- Initialize the object to the initializer PSA_KEY_DERIVATION_OPERATION_INIT, for example:

```
psa_key_derivation_operation_t operation = PSA_KEY_DERIVATION_OPERATION_
↪INIT;
```

- Assign the result of the function *psa_key_derivation_operation_init()* to the object, for example:

```
psa_key_derivation_operation_t operation;
operation = psa_key_derivation_operation_init();
```

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

3.2.1.5.8 typedef psa_mac_operation_t

type **psa_mac_operation_t**

The type of the state object for multi-part MAC operations.

3.2.1.5.8.1 Description

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

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

```
psa_mac_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_mac_operation_t operation;
```

- Initialize the object to the initializer PSA_MAC_OPERATION_INIT, for example:

```
psa_mac_operation_t operation = PSA_MAC_OPERATION_INIT;
```

- Assign the result of the function *psa_mac_operation_init()* to the object, for example:

```
psa_mac_operation_t operation;
operation = psa_mac_operation_init();
```

This is an implementation-defined type. Application should not make any assumptions about the content of this object.

3.2.1.5.9 PSA_CRYPTO_API_VERSION_MAJOR

The major version of this implementation of the PSA Crypto API.

3.2.1.5.10 PSA_CRYPTO_API_VERSION_MINOR

The minor version of this implementation of the PSA Crypto API.

3.2.1.5.11 PSA_KEY_DERIVATION_UNLIMITED_CAPACITY

Use the maximum possible capacity for a key derivation operation.

Use this value as the capacity argument when setting up a key derivation to specify that the operation will use the maximum possible capacity. The value of the maximum possible capacity depends on the key derivation algorithm.

3.2.1.5.12 psa_aead_abort

psa_status_t **psa_aead_abort**(*psa_aead_operation_t* *operation)

Abort an AEAD operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Initialized AEAD operation.

3.2.1.5.12.1 Description

Warning: Not supported

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_aead_encrypt_setup()* or *psa_aead_decrypt_setup()* again.

This function can be called any time after the operation object has been initialized as described in *typedef psa_aead_operation_t*.

In particular, calling *psa_aead_abort()* after the operation has been terminated by a call to *psa_aead_abort()*, *psa_aead_finish()* or *psa_aead_verify()* is safe and has no effect.

3.2.1.5.12.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.13 psa_aead_decrypt

psa_status_t **psa_aead_decrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *nonce, size_t nonce_length, const uint8_t *additional_data, size_t additional_data_length, const uint8_t *ciphertext, size_t ciphertext_length, uint8_t *plaintext, size_t plaintext_size, size_t *plaintext_length)

Process an authenticated decryption operation.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_DECRYPT.
- **alg** (*psa_algorithm_t*) – The AEAD algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
- **nonce** (const uint8_t*) – Nonce or IV to use.
- **nonce_length** (size_t) – Size of the nonce buffer in bytes. This must be appropriate for the selected algorithm. The default nonce size is PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type is the type of key.
- **additional_data** (const uint8_t*) – Additional data that has been authenticated but not encrypted.
- **additional_data_length** (size_t) – Size of additional_data in bytes.

- **ciphertext** (const uint8_t*) – Data that has been authenticated and encrypted. For algorithms where the encrypted data and the authentication tag are defined as separate inputs, the buffer must contain the encrypted data followed by the authentication tag.
- **ciphertext_length** (size_t) – Size of ciphertext in bytes.
- **plaintext** (uint8_t*) – Output buffer for the decrypted data.
- **plaintext_size** (size_t) – Size of the plaintext buffer in bytes.
- **plaintext_length** (size_t*) – On success, the size of the output in the plaintext buffer.

3.2.1.5.13.1 Description

Warning: Not supported

Parameter plaintext_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length) where key_type is the type of key.
- PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) evaluates to the maximum plaintext size of any supported AEAD decryption.

3.2.1.5.13.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_INVALID_SIGNATURE:**
  The ciphertext is not authentic.
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  key is not compatible with alg.
- **PSA_ERROR_NOT_SUPPORTED:**
  alg is not supported or is not an AEAD algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  plaintext_size is too small. *PSA_AEAD_DECRYPT_OUTPUT_SIZE()* or *PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE()* can be used to determine the required buffer size.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED

- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.14 psa_aead_decrypt_setup

psa_status_t **psa_aead_decrypt_setup**(*psa_aead_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set the key for a multi-part authenticated decryption operation.

Parameters

- **operation** (*psa_aead_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for *typedef psa_aead_operation_t* and not yet in use.
- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_DECRYPT.
- **alg** (*psa_algorithm_t*) – The AEAD algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.5.14.1 Description

Warning: Not supported

The sequence of operations to decrypt a message with authentication 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 *typedef psa_aead_operation_t*, e.g. PSA_AEAD_OPERATION_INIT.
3. Call *psa_aead_decrypt_setup()* to specify the algorithm and key.
4. If needed, call *psa_aead_set_lengths()* to specify the length of the inputs to the subsequent calls to *psa_aead_update_ad()* and *psa_aead_update()*. See the documentation of *psa_aead_set_lengths()* for details.
5. Call *psa_aead_set_nonce()* with the nonce for the decryption.
6. Call *psa_aead_update_ad()* zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call *psa_aead_update()* zero, one or more times, passing a fragment of the ciphertext to decrypt each time.
8. Call *psa_aead_verify()*.

If an error occurs at any step after a call to *psa_aead_decrypt_setup()*, the operation will need to be reset by a call to *psa_aead_abort()*. The application can call *psa_aead_abort()* at any time after the operation has been initialized.

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

- A successful call to *psa_aead_verify()*.
- A call to *psa_aead_abort()*.

3.2.1.5.14.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not an AEAD algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.15 psa_aead_encrypt

psa_status_t **psa_aead_encrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *nonce, size_t nonce_length, const uint8_t *additional_data, size_t additional_data_length, const uint8_t *plaintext, size_t plaintext_length, uint8_t *ciphertext, size_t ciphertext_size, size_t *ciphertext_length)

Process an authenticated encryption operation.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_ENCRYPT.
- **alg** (*psa_algorithm_t*) – The AEAD algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
- **nonce** (const uint8_t*) – Nonce or IV to use.
- **nonce_length** (size_t) – Size of the nonce buffer in bytes. This must be appropriate for the selected algorithm. The default nonce size is PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type is the type of key.
- **additional_data** (const uint8_t*) – Additional data that will be authenticated but not encrypted.
- **additional_data_length** (size_t) – Size of additional_data in bytes.
- **plaintext** (const uint8_t*) – Data that will be authenticated and encrypted.
- **plaintext_length** (size_t) – Size of plaintext in bytes.
- **ciphertext** (uint8_t*) – Output buffer for the authenticated and encrypted data. The additional data is not part of this output. For algorithms where the encrypted data and the authentication tag are defined as separate outputs, the authentication tag is appended to the encrypted data.
- **ciphertext_size** (size_t) – Size of the ciphertext buffer in bytes.
- **ciphertext_length** (size_t*) – On success, the size of the output in the ciphertext buffer.

3.2.1.5.15.1 Description

Warning: Not supported

Parameter ciphertext_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length) where key_type is the type of key.
- PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length) evaluates to the maximum ciphertext size of any supported AEAD encryption.

3.2.1.5.15.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_ENCRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not an AEAD algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  `ciphertext_size` is too small. *`PSA_AEAD_ENCRYPT_OUTPUT_SIZE()`* or *`PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE()`* can be used to determine the required buffer size.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *`psa_crypto_init()`*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.16 psa_aead_encrypt_setup

psa_status_t **`psa_aead_encrypt_setup`**(*psa_aead_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set the key for a multi-part authenticated encryption operation.

Parameters

- **`operation`** (*`psa_aead_operation_t`**) – The operation object to set up. It must have been initialized as per the documentation for *`typedef psa_aead_operation_t`* and not yet in use.
- **`key`** (*`psa_key_id_t`*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_ENCRYPT.
- **`alg`** (*`psa_algorithm_t`*) – The AEAD algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

3.2.1.5.16.1 Description

Warning: Not supported

The sequence of operations to encrypt a message with authentication is as follows:

- Allocate an operation object which will be passed to all the functions listed here.
- Initialize the operation object with one of the methods described in the documentation for *typedef psa_aead_operation_t*, e.g. PSA_AEAD_OPERATION_INIT.
- Call *psa_aead_encrypt_setup()* to specify the algorithm and key.
- If needed, call *psa_aead_set_lengths()* to specify the length of the inputs to the subsequent calls to *psa_aead_update_ad()* and *psa_aead_update()*. See the documentation of *psa_aead_set_lengths()* for details.
- Call either *psa_aead_generate_nonce()* or *psa_aead_set_nonce()* to generate or set the nonce. It is recommended to use *psa_aead_generate_nonce()* unless the protocol being implemented requires a specific nonce value.
- Call *psa_aead_update_ad()* zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
- Call *psa_aead_update()* zero, one or more times, passing a fragment of the message to encrypt each time.
- Call *psa_aead_finish()*.

If an error occurs at any step after a call to *psa_aead_encrypt_setup()*, the operation will need to be reset by a call to *psa_aead_abort()*. The application can call *psa_aead_abort()* at any time after the operation has been initialized.

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

- A successful call to *psa_aead_finish()*.
- A call to *psa_aead_abort()*.

3.2.1.5.16.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_ENCRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not an AEAD algorithm.

- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.17 psa_aead_finish

psa_status_t **psa_aead_finish**(*psa_aead_operation_t* *operation, uint8_t *ciphertext, size_t ciphertext_size, size_t *ciphertext_length, uint8_t *tag, size_t tag_size, size_t *tag_length)

Finish encrypting a message in an AEAD operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **ciphertext** (uint8_t*) – Buffer where the last part of the ciphertext is to be written.
- **ciphertext_size** (size_t) – Size of the ciphertext buffer in bytes.
- **ciphertext_length** (size_t*) – On success, the number of bytes of returned ciphertext.
- **tag** (uint8_t*) – Buffer where the authentication tag is to be written.
- **tag_size** (size_t) – Size of the tag buffer in bytes.
- **tag_length** (size_t*) – On success, the number of bytes that make up the returned tag.

3.2.1.5.17.1 Description

Warning: Not supported

The operation must have been set up with *psa_aead_encrypt_setup()*.

This function finishes the authentication of the additional data formed by concatenating the inputs passed to preceding calls to *psa_aead_update_ad()* with the plaintext formed by concatenating the inputs passed to preceding calls to *psa_aead_update()*.

This function has two output buffers:

- ciphertext contains trailing ciphertext that was buffered from preceding calls to *psa_aead_update()*.

- `tag` contains the authentication tag.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_aead_abort()*.

Parameter `ciphertext_size` must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation.
- PSA_AEAD_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.

Parameter `tag_size` must be appropriate for the selected algorithm and key:

- The exact tag size is PSA_AEAD_TAG_LENGTH(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size of the key, and alg is the algorithm that were used in the call to *psa_aead_encrypt_setup()*.
- PSA_AEAD_TAG_MAX_SIZE evaluates to the maximum tag size of any supported AEAD algorithm.

3.2.1.5.17.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE**
  The operation state is not valid: it must be an active encryption operation with a nonce set.
- **PSA_ERROR_BUFFER_TOO_SMALL**
  The size of the `ciphertext` or `tag` buffer is too small. *PSA_AEAD_FINISH_OUTPUT_SIZE()* or PSA_AEAD_FINISH_OUTPUT_MAX_SIZE can be used to determine the required ciphertext buffer size. *PSA_AEAD_TAG_LENGTH()* or PSA_AEAD_TAG_MAX_SIZE can be used to determine the required `tag` buffer size.
- **PSA_ERROR_INVALID_ARGUMENT**
  The total length of input to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.
- **PSA_ERROR_INVALID_ARGUMENT**
  The total length of input to *psa_aead_update()* so far is less than the plaintext length that was previously specified with *psa_aead_set_lengths()*.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID

- **PSA_ERROR_BAD_STATE**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.18 psa_aead_generate_nonce

psa_status_t **psa_aead_generate_nonce**(*psa_aead_operation_t* *operation, uint8_t *nonce, size_t nonce_size, size_t *nonce_length)

Generate a random nonce for an authenticated encryption operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **nonce** (uint8_t*) – Buffer where the generated nonce is to be written.
- **nonce_size** (size_t) – Size of the nonce buffer in bytes. This must be at least PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type and alg are type of key and the algorithm respectively that were used to set up the AEAD operation.
- **nonce_length** (size_t*) – On success, the number of bytes of the generated nonce.

3.2.1.5.18.1 Description

Warning: Not supported

This function generates a random nonce for the authenticated encryption operation with an appropriate size for the chosen algorithm, key type and key size.

The application must call *psa_aead_encrypt_setup()* before calling this function. If applicable for the algorithm, the application must call *psa_aead_set_lengths()* before calling this function.

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

3.2.1.5.18.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be an active AEAD encryption operation, with no nonce set.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: this is an algorithm which requires *psa_aead_set_lengths()* to be called before setting the nonce.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the nonce buffer is too small. *PSA_AEAD_NONCE_LENGTH()* or PSA_AEAD_NONCE_MAX_SIZE can be used to determine the required buffer size.

- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.19 psa_aead_operation_init

psa_aead_operation_t **psa_aead_operation_init**(void)

Return an initial value for an AEAD operation object.

Parameters

- **void** – no arguments

3.2.1.5.19.1 Description

Warning: Not supported

3.2.1.5.19.2 Return

typedef psa_aead_operation_t

3.2.1.5.20 psa_aead_set_lengths

psa_status_t **psa_aead_set_lengths**(*psa_aead_operation_t* *operation, size_t ad_length, size_t plaintext_length)

Declare the lengths of the message and additional data for AEAD.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **ad_length** (size_t) – Size of the non-encrypted additional authenticated data in bytes.
- **plaintext_length** (size_t) – Size of the plaintext to encrypt in bytes.

3.2.1.5.20.1 Description

Warning: Not supported

The application must call this function before calling *psa_aead_set_nonce()* or *psa_aead_generate_nonce()*, if the algorithm for the operation requires it. If the algorithm does not require it, calling this function is optional, but if this function is called then the implementation must enforce the lengths.

- For PSA_ALG_CCM, calling this function is required.
- For the other AEAD algorithms defined in this specification, calling this function is not required.
- For vendor-defined algorithm, refer to the vendor documentation.

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

3.2.1.5.20.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, and *psa_aead_set_nonce()* and *psa_aead_generate_nonce()* must not have been called yet.
- **PSA_ERROR_INVALID_ARGUMENT:**
  At least one of the lengths is not acceptable for the chosen algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.21 psa_aead_set_nonce

psa_status_t **psa_aead_set_nonce**(*psa_aead_operation_t* *operation, const uint8_t *nonce, size_t nonce_length)

Set the nonce for an authenticated encryption or decryption operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **nonce** (const uint8_t*) – Buffer containing the nonce to use.
- **nonce_length** (size_t) – Size of the nonce in bytes. This must be a valid nonce size for the chosen algorithm. The default nonce size is PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type and alg are

type of key and the algorithm respectively that were used to set up the AEAD operation.

3.2.1.5.21.1 Description

This function sets the nonce for the authenticated encryption or decryption operation.

Warning: Not supported

The application must call *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()* before calling this function. If applicable for the algorithm, the application must call *psa_aead_set_lengths()* before calling this function.

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

Note:
When encrypting, *psa_aead_generate_nonce()* is recommended instead of using this function, unless implementing a protocol that requires a non-random IV.

3.2.1.5.21.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, with no nonce set.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: this is an algorithm which requires *psa_aead_set_lengths()* to be called before setting the nonce.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The size of nonce is not acceptable for the chosen algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.22 psa_aead_update

psa_status_t **psa_aead_update**(*psa_aead_operation_t* *operation, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypt or decrypt a message fragment in an active AEAD operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **input** (const uint8_t*) – Buffer containing the message fragment to encrypt or decrypt.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **output** (uint8_t*) – Buffer where the output is to be written.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the returned output.

3.2.1.5.22.1 Description

Warning: Not supported

The following must occur before calling this function:

- Call either *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()*. The choice of setup function determines whether this function encrypts or decrypts its input.
- Set the nonce with *psa_aead_generate_nonce()* or *psa_aead_set_nonce()*.
- Call *psa_aead_update_ad()* to pass all the additional data.

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

This function does not require the input to be aligned to any particular block boundary. If the implementation can only process a whole block at a time, it must consume all the input provided, but it might delay the end of the corresponding output until a subsequent call to *psa_aead_update()*, *psa_aead_finish()* or *psa_aead_verify()* provides sufficient input. The amount of data that can be delayed in this way is bounded by *PSA_AEAD_UPDATE_OUTPUT_SIZE()*.

Parameter output_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_AEAD_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of key and alg is the algorithm that were used to set up the operation.
- PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported AEAD algorithm.

3.2.1.5.22.2 Return

- **PSA_SUCCESS:**
  Success.

  Warning:
  When decrypting, do not use the output until *psa_aead_verify()* succeeds.

  See the detailed warning.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, have a nonce set, and have lengths set if required by the algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `output` buffer is too small. *PSA_AEAD_UPDATE_OUTPUT_SIZE()* or *PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE()* can be used to determine the required buffer size.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total length of `input` to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total input length overflows the plaintext length that was previously specified with *psa_aead_set_lengths()*.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.23 psa_aead_update_ad

psa_status_t **psa_aead_update_ad**(*psa_aead_operation_t* *operation, const uint8_t *input, size_t input_length)

Pass additional data to an active AEAD operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **input** (const uint8_t*) – Buffer containing the fragment of additional data.
- **input_length** (size_t) – S ize of the `input` buffer in bytes.

3.2.1.5.23.1 Description

Warning: Not supported

Additional data is authenticated, but not encrypted.

This function can be called multiple times to pass successive fragments of the additional data. This function must not be called after passing data to encrypt or decrypt with *psa_aead_update()*.

3.2.1.5.23.2 The following must occur before calling this function

- Call either *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()*.
- Set the nonce with *psa_aead_generate_nonce()* or *psa_aead_set_nonce()*.

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

3.2.1.5.23.3 Return

- **PSA_SUCCESS:**
  Success.

  Warning:
  When decrypting, do not trust the input until *psa_aead_verify()* succeeds.

  See the detailed warning.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, have a nonce set, have lengths set if required by the algorithm, and *psa_aead_update()* must not have been called yet.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total input length overflows the additional data length that was previously specified with *psa_aead_set_lengths()*.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.24 psa_aead_verify

psa_status_t **psa_aead_verify**(*psa_aead_operation_t* *operation, uint8_t *plaintext, size_t plaintext_size, size_t *plaintext_length, const uint8_t *tag, size_t tag_length)

Finish authenticating and decrypting a message in an AEAD operation.

Parameters

- **operation** (*psa_aead_operation_t**) – Active AEAD operation.
- **plaintext** (uint8_t*) – Buffer where the last part of the plaintext is to be written. This is the remaining data from previous calls to `psa_aead_update()` that could not be processed until the end of the input.
- **plaintext_size** (size_t) – Size of the `plaintext` buffer in bytes.
- **plaintext_length** (size_t*) – On success, the number of bytes of returned plaintext.
- **tag** (const uint8_t*) – Buffer containing the authentication tag.
- **tag_length** (size_t) – Size of the `tag` buffer in bytes.

3.2.1.5.24.1 Description

Warning: Not supported

The operation must have been set up with `psa_aead_decrypt_setup()`.

This function finishes the authenticated decryption of the message components:

- The additional data consisting of the concatenation of the inputs passed to preceding calls to `psa_aead_update_ad()`.
- The ciphertext consisting of the concatenation of the inputs passed to preceding calls to `psa_aead_update()`.
- The tag passed to this function call.

If the authentication tag is correct, this function outputs any remaining plaintext and reports success. If the authentication tag is not correct, this function returns PSA_ERROR_INVALID_SIGNATURE.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_aead_abort()`.

Note:

Implementations must make the best effort to ensure that the comparison between the actual tag and the expected tag is performed in constant time.

Parameter `plaintext_size` must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation.
- PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.

3.2.1.5.24.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The calculations were successful, but the authentication tag is not correct.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be an active decryption operation with a nonce set.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `plaintext` buffer is too small. *PSA_AEAD_VERIFY_OUTPUT_SIZE()* or PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total length of input to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total length of input to *psa_aead_update()* so far is less than the plaintext length that was previously specified with *psa_aead_set_lengths()*.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.25 psa_asymmetric_decrypt

psa_status_t **psa_asymmetric_decrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *salt, size_t salt_length, uint8_t *output, size_t output_size, size_t *output_length)

Decrypt a short message with a private key.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must be an asymmetric key pair. It must allow the usage PSA_KEY_USAGE_DECRYPT.
- **alg** (*psa_algorithm_t*) – An asymmetric encryption algorithm that is compatible with the type of key.

- **input** (const uint8_t*) – The message to decrypt.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **salt** (const uint8_t*) – A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass NULL. If the algorithm supports an optional salt, pass NULL to indicate that there is no salt.
- **salt_length** (size_t) – Size of the salt buffer in bytes. If salt is NULL, pass 0.
- **output** (uint8_t*) – Buffer where the decrypted message is to be written.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the returned output.

3.2.1.5.25.1 Description

Warning: Not supported

For PSA_ALG_RSA_PKCS1V15_CRYPT, no salt is supported.

Parameter output_size must be appropriate for the selected algorithm and key:

- The required output size is PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of key.
- PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported asymmetric decryption.

3.2.1.5.25.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the output buffer is too small. *PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE()* or PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED

- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_ENTROPY
- PSA_ERROR_INVALID_PADDING
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.26 psa_asymmetric_encrypt

psa_status_t **psa_asymmetric_encrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *salt, size_t salt_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypt a short message with a public key.

Parameters

- **key** (*psa_key_id_t*) – Identifer of the key to use for the operation. It must be a public key or an asymmetric key pair. It must allow the usage PSA_KEY_USAGE_ENCRYPT.
- **alg** (*psa_algorithm_t*) – An asymmetric encryption algorithm that is compatible with the type of key.
- **input** (`const uint8_t*`) – The message to encrypt.
- **input_length** (`size_t`) – Size of the `input` buffer in bytes.
- **salt** (`const uint8_t*`) – A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass NULL. If the algorithm supports an optional salt, pass NULL to indicate that there is no salt.
- **salt_length** (`size_t`) – Size of the `salt` buffer in bytes. If salt is NULL, pass 0.
- **output** (`uint8_t*`) – Buffer where the encrypted message is to be written.
- **output_size** (`size_t`) – Size of the `output` buffer in bytes.
- **output_length** (`size_t*`) – On success, the number of bytes that make up the returned output.

3.2.1.5.26.1 Description

Warning: Not supported

For PSA_ALG_RSA_PKCS1V15_CRYPT, no salt is supported.

Parameter `output_size` must be appropriate for the selected algorithm and key:

- The required output size is PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of `key`.
- PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported asymmetric encryption.

3.2.1.5.26.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_ENCRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `output` buffer is too small. *`PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE()`* or PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_ENTROPY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *`psa_crypto_init()`*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.27 psa_cipher_abort

psa_status_t **psa_cipher_abort**(*psa_cipher_operation_t* *operation)

Abort a cipher operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – Initialized cipher operation.

3.2.1.5.27.1 Description

Warning: Not supported

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_cipher_encrypt_setup()` or `psa_cipher_decrypt_setup()` again.

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

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

3.2.1.5.27.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.28 psa_cipher_decrypt

psa_status_t **psa_cipher_decrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Decrypt a message using a symmetric cipher.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_DECRYPT.
- **alg** (*psa_algorithm_t*) – The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

- **input** (const uint8_t*) – Buffer containing the message to decrypt. This consists of the IV followed by the ciphertext proper.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **output** (uint8_t*) – Buffer where the plaintext is to be written.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the output.

3.2.1.5.28.1 Description

This function decrypts a message encrypted with a symmetric cipher.

The input to this function must contain the IV followed by the ciphertext, as output by *psa_cipher_encrypt()*. The IV must be PSA_CIPHER_IV_LENGTH(key_type, alg) bytes in length, where key_type is the type of key.

Use the multi-part operation interface with a *typedef psa_cipher_operation_t* object to decrypt data which is not in the expected input format.

Parameter output_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of key.
- PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported cipher decryption.

3.2.1.5.28.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  key is not compatible with alg.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The input_length is not valid for the algorithm and key type. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
- **PSA_ERROR_NOT_SUPPORTED:**
  alg is not supported or is not a cipher algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  output_size is too small. *PSA_CIPHER_DECRYPT_OUTPUT_SIZE()* or *PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE()* can be used to determine the required buffer size.

- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.29 psa_cipher_decrypt_setup

psa_status_t **psa_cipher_decrypt_setup**(*psa_cipher_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set the key for a multi-part symmetric decryption operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for *typedef psa_cipher_operation_t* and not yet in use.
- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_DECRYPT.
- **alg** (*psa_algorithm_t*) – The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

3.2.1.5.29.1 Description

Warning: Not supported

The sequence of operations to decrypt a message with a symmetric cipher is as follows:

- Allocate an operation object which will be passed to all the functions listed here.
- Initialize the operation object with one of the methods described in the documentation for *typedef psa_cipher_operation_t*, e.g. PSA_CIPHER_OPERATION_INIT.
- Call *psa_cipher_decrypt_setup()* to specify the algorithm and key.
- Call *psa_cipher_set_iv()* with the initialization vector (IV) for the decryption, if the algorithm requires one. This must match the IV used for the encryption.
- Call *psa_cipher_update()* zero, one or more times, passing a fragment of the message each time.
- Call *psa_cipher_finish()*.

If an error occurs at any step after a call to *psa_cipher_decrypt_setup()*, the operation will need to be reset by a call to *psa_cipher_abort()*. The application can call *psa_cipher_abort()* at any time after the operation has been initialized.

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

- A successful call to *psa_cipher_finish()*.
- A call to *psa_cipher_abort()*.

3.2.1.5.29.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a cipher algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.30 psa_cipher_encrypt

psa_status_t **psa_cipher_encrypt**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypt a message using a symmetric cipher.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_ENCRYPT.
- **alg** (*psa_algorithm_t*) – The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
- **input** (const uint8_t*) – Buffer containing the message to encrypt.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **output** (uint8_t*) – Buffer where the output is to be written. The output contains the IV followed by the ciphertext proper.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the output.

3.2.1.5.30.1 Description

This function encrypts a message with a random initialization vector (IV). The length of the IV is PSA_CIPHER_IV_LENGTH(key_type, alg) where key_type is the type of key. The output of *psa_cipher_encrypt()* is the IV followed by the ciphertext.

Use the multi-part operation interface with a *typedef psa_cipher_operation_t* object to provide other forms of IV or to manage the IV and ciphertext independently.

Parameter output_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of key.
- PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported cipher encryption.

3.2.1.5.30.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_ENCRYPT flag, or it does not permit the requested algorithm.

- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The input_length is not valid for the algorithm and key type. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a cipher algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  `output_size` is too small. *`PSA_CIPHER_ENCRYPT_OUTPUT_SIZE()`* or *`PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE()`* can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *`psa_crypto_init()`*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.31 psa_cipher_encrypt_setup

psa_status_t **`psa_cipher_encrypt_setup`**(*psa_cipher_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set the key for a multi-part symmetric encryption operation.

Parameters

- **`operation`** (*`psa_cipher_operation_t`**) – The operation object to set up. It must have been initialized as per the documentation for *`typedef psa_cipher_operation_t`* and not yet in use.
- **`key`** (*`psa_key_id_t`*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_ENCRYPT.
- **`alg`** (*`psa_algorithm_t`*) – The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

3.2.1.5.31.1 Description

Warning: Not supported

The sequence of operations to encrypt a message with a symmetric cipher is as follows:

- Allocate an operation object which will be passed to all the functions listed here.
- Initialize the operation object with one of the methods described in the documentation for *typedef psa_cipher_operation_t*, e.g. PSA_CIPHER_OPERATION_INIT.
- Call *psa_cipher_encrypt_setup()* to specify the algorithm and key.
- Call either *psa_cipher_generate_iv()* or *psa_cipher_set_iv()* to generate or set the initialization vector (IV), if the algorithm requires one. It is recommended to use *psa_cipher_generate_iv()* unless the protocol being implemented requires a specific IV value.
- Call *psa_cipher_update()* zero, one or more times, passing a fragment of the message each time.
- Call *psa_cipher_finish()*.

If an error occurs at any step after a call to *psa_cipher_encrypt_setup()*, the operation will need to be reset by a call to *psa_cipher_abort()*. The application can call *psa_cipher_abort()* at any time after the operation has been initialized.

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

- A successful call to *psa_cipher_finish()*.
- A call to *psa_cipher_abort()*.

3.2.1.5.31.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_ENCRYPT flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a cipher algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE

- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.32 psa_cipher_finish

psa_status_t **psa_cipher_finish**(*psa_cipher_operation_t* *operation, uint8_t *output, size_t output_size, size_t *output_length)

Finish encrypting or decrypting a message in a cipher operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – Active cipher operation.
- **output** (uint8_t*) – Buffer where the output is to be written.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the returned output.

3.2.1.5.32.1 Description

Warning: Not supported

The application must call *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()* before calling this function. The choice of setup function determines whether this function encrypts or decrypts its input.

This function finishes the encryption or decryption of the message formed by concatenating the inputs passed to preceding calls to *psa_cipher_update()*.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_cipher_abort()*.

Parameter output_size must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation.
- PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported cipher algorithm.

3.2.1.5.32.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The total input size passed to this operation is not valid for this particular algorithm. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
- **PSA_ERROR_INVALID_PADDING:**
  This is a decryption operation for an algorithm that includes padding, and the ciphertext does not contain valid padding.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, with an IV set if required for the algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `output` buffer is too small. *PSA_CIPHER_FINISH_OUTPUT_SIZE()* or PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.33 psa_cipher_generate_iv

psa_status_t **psa_cipher_generate_iv**(*psa_cipher_operation_t* *operation, uint8_t *iv, size_t iv_size, size_t *iv_length)

Generate an initialization vector (IV) for a symmetric encryption operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – Active cipher operation.
- **iv** (uint8_t*) – Buffer where the generated IV is to be written.
- **iv_size** (size_t) – Size of the `iv` buffer in bytes. This must be at least PSA_CIPHER_IV_LENGTH(key_type, alg) where key_type and alg are type of key and the algorithm respectively that were used to set up the cipher operation.
- **iv_length** (size_t*) – On success, the number of bytes of the generated IV.

3.2.1.5.33.1 Description

Warning: Not supported

This function generates a random IV, nonce or initial counter value for the encryption operation as appropriate for the chosen algorithm, key type and key size.

The generated IV is always the default length for the key and algorithm: PSA_CIPHER_IV_LENGTH(key_type, alg), where key_type is the type of `key` and alg is the algorithm that were used to set up the operation. To generate different lengths of IV, use *psa_generate_random()* and *psa_cipher_set_iv()*.

If the cipher algorithm does not use an IV, calling this function returns a PSA_ERROR_BAD_STATE error. For these algorithms, PSA_CIPHER_IV_LENGTH(key_type, alg) will be zero.

The application must call *psa_cipher_encrypt_setup()* before calling this function.

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

3.2.1.5.33.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  Either:
  - The cipher algorithm does not use an IV.
  - The operation state is not valid: it must be active, with no IV set.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `iv` buffer is too small. *PSA_CIPHER_IV_LENGTH()* or PSA_CIPHER_IV_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.34 psa_cipher_operation_init

psa_cipher_operation_t **psa_cipher_operation_init**(void)

Return an initial value for a cipher operation object.

Parameters

- **void** – no arguments

3.2.1.5.34.1 Return

typedef psa_cipher_operation_t

3.2.1.5.35 psa_cipher_set_iv

psa_status_t **psa_cipher_set_iv**(*psa_cipher_operation_t* *operation, const uint8_t *iv, size_t iv_length)

Set the initialization vector (IV) for a symmetric encryption or decryption operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – Active cipher operation.
- **iv** (const uint8_t*) – Buffer containing the IV to use.
- **iv_length** (size_t) – Size of the IV in bytes.

3.2.1.5.35.1 Description

Warning: Not supported

This function sets the IV, nonce or initial counter value for the encryption or decryption operation.

If the cipher algorithm does not use an IV, calling this function returns a PSA_ERROR_BAD_STATE error. For these algorithms, PSA_CIPHER_IV_LENGTH(key_type, alg) will be zero.

The application must call *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()* before calling this function.

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

Note:

When encrypting, *psa_cipher_generate_iv()* is recommended instead of using this function, unless implementing a protocol that requires a non-random IV.

3.2.1.5.35.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  Either:
  - The cipher algorithm does not use an IV.
  - The operation state is not valid: it must be an active cipher encrypt operation, with no IV set.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The size of iv is not acceptable for the chosen algorithm, or the chosen algorithm does not use an IV.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.36 psa_cipher_update

psa_status_t **psa_cipher_update**(*psa_cipher_operation_t* *operation, const uint8_t *input, size_t input_length, uint8_t *output, size_t output_size, size_t *output_length)

Encrypt or decrypt a message fragment in an active cipher operation.

Parameters

- **operation** (*psa_cipher_operation_t**) – Active cipher operation.
- **input** (const uint8_t*) – Buffer containing the message fragment to encrypt or decrypt.
- **input_length** (size_t) – Size of the `input` buffer in bytes.
- **output** (uint8_t*) – Buffer where the output is to be written.
- **output_size** (size_t) – Size of the `output` buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the returned output.

3.2.1.5.36.1 Description

Warning: Not supported

The following must occur before calling this function:

1. Call either *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()*. The choice of setup function determines whether this function encrypts or decrypts its input.
2. If the algorithm requires an IV, call *psa_cipher_generate_iv()* or *psa_cipher_set_iv()*. *psa_cipher_generate_iv()* is recommended when encrypting.

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

Parameter `output_size` must be appropriate for the selected algorithm and key:

- A sufficient output size is PSA_CIPHER_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of `key` and alg is the algorithm that were used to set up the operation.
- PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported cipher algorithm.

3.2.1.5.36.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active, with an IV set if required for the algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `output` buffer is too small. *PSA_CIPHER_UPDATE_OUTPUT_SIZE()* or *PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE()* can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.37 psa_copy_key

psa_status_t **psa_copy_key**(*psa_key_id_t* source_key, const *psa_key_attributes_t* *attributes, *psa_key_id_t* *target_key)

Make a copy of a key.

Parameters

- **source_key** (*psa_key_id_t*) – The key to copy. It must allow the usage PSA_KEY_USAGE_COPY. If a private or secret key is being copied outside of a secure element it must also allow PSA_KEY_USAGE_EXPORT.
- **attributes** (const *psa_key_attributes_t**) – The attributes for the new key.
- **target_key** (*psa_key_id_t**) – On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

3.2.1.5.37.1 Description

Warning: Not supported

Copy key material from one location to another.

This function is primarily useful to copy a key from one location to another, as it populates a key using the material from another key which can have a different lifetime.

This function can be used to share a key with a different party.

The policy on the source key must have the usage flag PSA_KEY_USAGE_COPY set. This flag is sufficient to permit the copy if the key has the lifetime PSA_KEY_LIFETIME_VOLATILE or PSA_KEY_LIFETIME_PERSISTENT. Some secure elements do not provide a way to copy a key without making it extractable from the secure element. If a key is located in such a secure element,then the key must have both usage flags PSA_KEY_USAGE_COPY and PSA_KEY_USAGE_EXPORT in order to make a copy of the key outside the secure element.

The resulting key can only be used in a way that conforms to both the policy of the original key and the policy specified in the attributes parameter:

- The usage flags on the resulting key are the bitwise-and of the usage flags on the source policy and the usage flags in attributes.
- If both permit the same algorithm or wildcard-based algorithm, the resulting key has the same permitted algorithm.
- If either of the policies permits an algorithm and the other policy allows a wildcard-based permitted algorithm that includes this algorithm, the resulting key uses this permitted algorithm.
- If the policies do not permit any algorithm in common, this function fails with the status PSA_ERROR_INVALID_ARGUMENT.

The effect of this function on implementation-defined attributes is implementation-defined.

This function uses the attributes as follows:

- The key type and size can be 0. If either is nonzero, it must match the corresponding attribut of the source key.

- The key location (the lifetime and, for persistent keys, the key identifier) is used directly.
- The key policy (usage flags and permitted algorithm) are combined from the source key and attributes so that both sets of restrictions apply, as described in the documentation of this function.

  Note:

  This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling *psa_get_key_attributes()* with the key's identifier.

3.2.1.5.37.2 Return

- **PSA_SUCCESS:**
  Success. If the new key is persistent, the key material and the key's metadata have been saved to persistent storage.
- **PSA_ERROR_INVALID_HANDLE:**
  `source_key` is invalid.
- **PSA_ERROR_ALREADY_EXISTS:**
  This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The lifetime or identifier in `attributes` are invalid.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key policies from `source_key` and specified in `attributes` are incompatible.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `attributes` specifies a key type or key size which does not match the attributes of source key.
- **PSA_ERROR_NOT_PERMITTED:**
  `source_key` does not have the PSA_KEY_USAGE_COPY usage flag.
- **PSA_ERROR_NOT_PERMITTED:**
  `source_key` does not have the PSA_KEY_USAGE_EXPORT usage flag and its lifetime does not allow copying it to the target's lifetime.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_INSUFFICIENT_STORAGE
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.38 psa_crypto_init

psa_status_t **psa_crypto_init**(void)

Library initialization.

Parameters

- **void** – no arguments

3.2.1.5.38.1 Description

Applications must call this function before calling any other function in this module.

Applications are permitted to call this function more than once. Once a call succeeds, subsequent calls are guaranteed to succeed.

If the application calls other functions before calling *psa_crypto_init()*, the behavior is undefined. In this situation:

- Implementations are encouraged to either perform the operation as if the library had been initialized or to return PSA_ERROR_BAD_STATE or some other applicable error.
- Implementations must not return a success status if the lack of initialization might have security implications, for example due to improper seeding of the random number generator.

3.2.1.5.38.2 Return

- PSA_SUCCESS
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_INSUFFICIENT_ENTROPY

3.2.1.5.39 psa_destroy_key

psa_status_t **psa_destroy_key**(*psa_key_id_t* key)

Destroy a key.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to erase. If this is PSA_KEY_ID_NULL, do nothing and return PSA_SUCCESS.

3.2.1.5.39.1 Description

This function destroys a key from both volatile memory and, if applicable, non-volatile storage. Implementations must make a best effort to ensure that the key material cannot be recovered.

This function also erases any metadata such as policies and frees resources associated with the key.

Destroying the key makes the key identifier invalid, and the key identifier must not be used again by the application.

If a key is currently in use in a multi-part operation, then destroying the key will cause the multi-part operation to fail.

3.2.1.5.39.2 Return

- **PSA_SUCCESS:**
  `key` was a valid key identifier and the key material that it referred to has been erased. Alternatively, key is PSA_KEY_ID_NULL.
- **PSA_ERROR_NOT_PERMITTED:**
  The key cannot be erased because it is read-only, either due to a policy or due to physical restrictions.
- **PSA_ERROR_INVALID_HANDLE:**
  `key` is not a valid handle nor PSA_KEY_ID_NULL.
- **PSA_ERROR_COMMUNICATION_FAILURE**
  There was an failure in communication with the cryptoprocessor. The key material might still be present in the cryptoprocessor.
- **PSA_ERROR_STORAGE_FAILURE:**
  The storage operation failed. Implementations must make a best effort to erase key material even in this situation, however, it might be impossible to guarantee that the key material is not recoverable in such cases.
- **PSA_ERROR_DATA_CORRUPT:**
  The storage is corrupted. Implementations must make a best effort to erase key material even in this situation, however, it might be impossible to guarantee that the key material is not recoverable in such cases.
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_CORRUPTION_DETECTED:**
  An unexpected condition which is not a storage corruption or a communication failure occurred. The cryptoprocessor might have been compromised.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.40 psa_export_key

psa_status_t **psa_export_key**(*psa_key_id_t* key, uint8_t *data, size_t data_size, size_t *data_length)

Export a key in binary format.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to export. It must allow the usage PSA_KEY_USAGE_EXPORT, unless it is a public key.
- **data** (uint8_t*) – Buffer where the key data is to be written.
- **data_size** (size_t) – Size of the data buffer in bytes.
- **data_length** (size_t*) – On success, the number of bytes that make up the key data.

3.2.1.5.40.1 Description

Warning: Export of any private key is not supported for now.

The output of this function can be passed to *psa_import_key()* to create an equivalent object.

If the implementation of *psa_import_key()* supports other formats beyond the format specified here, the output from *psa_export_key()* must use the representation specified here, not the original representation.

For standard key types, the output format is as follows:

- For symmetric keys, excluding HMAC keys, the format is the raw bytes of the key.
- For HMAC keys that are shorter than, or equal in size to, the underlying hash algorithm block size, the format is the raw bytes of the key.

  For HMAC keys that are longer than the underlying hash algorithm block size, the format is an implementation defined choice between the following formats:

  1. The raw bytes of the key.
  2. The raw bytes of the hash of the key, using the underlying hash algorithm.

  See also PSA_KEY_TYPE_HMAC.

- For DES, the key data consists of 8 bytes. The parity bits must be correct.
- For Triple-DES, the format is the concatenation of the two or three DES keys.
- For RSA key pairs, with key type PSA_KEY_TYPE_RSA_KEY_PAIR, the format is the non-encrypted DER encoding of the representation defined by in PKCS #1: RSA Cryptography Specifications Version 2.2 [RFC8017] as RSAPrivateKey, version 0.

```
RSAPrivateKey ::= SEQUENCE {
    version             INTEGER,  -- must be 0
    modulus             INTEGER,  -- n
    publicExponent      INTEGER,  -- e
    privateExponent     INTEGER,  -- d
    prime1              INTEGER,  -- p
    prime2              INTEGER,  -- q
```

(continues on next page)

(continued from previous page)

```
    exponent1           INTEGER,  -- d mod (p-1)
    exponent2           INTEGER,  -- d mod (q-1)
    coefficient         INTEGER,  -- (inverse of q) mod p
}
```

Note:

Although it is possible to define an RSA key pair or private key using a subset of these elements, the output from psa_export_key() for an RSA key pair must include all of these elements.

- For elliptic curve key pairs, with key types for which *PSA_KEY_TYPE_IS_ECC_KEY_PAIR()* is true, the format is a representation of the private value.
  - For Weierstrass curve families PSA_ECC_FAMILY_SECT_XX, PSA_ECC_FAMILY_SECP_XX, PSA_ECC_FAMILY_FRP and PSA_ECC_FAMILY_BRAINPOOL_P_R1, the content of the privateKey field of the ECPrivateKey format defined by Elliptic Curve Private Key Structure [RFC5915].

    This is a ceiling(m/8)-byte string in big-endian order where m is the key size in bits.
  - For curve family PSA_ECC_FAMILY_MONTGOMERY, the scalar value of the 'private key' in little-endian order as defined by Elliptic Curves for Security [RFC7748] §6. The value must have the forced bits set to zero or one as specified by `decodeScalar25519()` and `decodeScalar448()` in [RFC7748] §5.

    This is a ceiling(m/8)-byte string where m is the key size in bits. This is 32 bytes for Curve25519, and 56 bytes for Curve448.
- For Diffie-Hellman key exchange key pairs, with key types for which *PSA_KEY_TYPE_IS_DH_KEY_PAIR()* is true, the format is the representation of the private key x as a big-endian byte string. The length of the byte string is the private key size in bytes, and leading zeroes are not stripped.
- For public keys, with key types for which *PSA_KEY_TYPE_IS_PUBLIC_KEY()* is true, the format is the same as for *psa_export_public_key()*.

The policy on the key must have the usage flag PSA_KEY_USAGE_EXPORT set.

Parameter `data_size` must be appropriate for the key:

- The required output size is PSA_EXPORT_KEY_OUTPUT_SIZE(type, bits) where type is the key type and bits is the key size in bits.
- PSA_EXPORT_KEY_PAIR_MAX_SIZE evaluates to the maximum output size of any supported key pair.
- PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key.
- This API defines no maximum size for symmetric keys. Arbitrarily large data items can be stored in the key store, for example certificates that correspond to a stored private key or input material for key derivation.

3.2.1.5.40.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_EXPORT flag.
- PSA_ERROR_NOT_SUPPORTED
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `data` buffer is too small. *PSA_EXPORT_KEY_OUTPUT_SIZE()* or PSA_EXPORT_KEY_PAIR_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.41 psa_export_public_key

psa_status_t **psa_export_public_key**(*psa_key_id_t* key, uint8_t *data, size_t data_size, size_t *data_length)

Export a public key or the public part of a key pair in binary format.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to export.
- **data** (`uint8_t*`) – Buffer where the key data is to be written.
- **data_size** (`size_t`) – Size of the `data` buffer in bytes.
- **data_length** (`size_t*`) – On success, the number of bytes that make up the key data.

3.2.1.5.41.1 Description

Warning: Not supported

The output of this function can be passed to *psa_import_key()* to create an object that is equivalent to the public key.

If the implementation of *psa_import_key()* supports other formats beyond the format specified here, the output from *psa_export_public_key()* must use the representation specified here, not the original representation.

For standard key types, the output format is as follows:

- For RSA public keys, with key type PSA_KEY_TYPE_RSA_PUBLIC_KEY, the DER encoding of the representation defined by Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [RFC3279] §2.3.1 as RSAPublicKey.

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER,    -- n
   publicExponent     INTEGER  }  -- e
```

- For elliptic curve key pairs, with key types for which *PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY()* is true, the format depends on the key family:
  - For Weierstrass curve families PSA_ECC_FAMILY_SECT_XX, PSA_ECC_FAMILY_SECP_XX, PSA_ECC_FAMILY_FRP and PSA_ECC_FAMILY_BRAINPOOL_P_R1, the uncompressed representation of an elliptic curve point as an octet string defined in SEC 1: Elliptic Curve Cryptography [SEC1] §2.3.3. If m is the bit size associated with the curve, i.e. the bit size of q for a curve over F_q. The representation consists of:
    - The byte 0x04;
    - x_P as a ceiling(m/8)-byte string, big-endian;
    - y_P as a ceiling(m/8)-byte string, big-endian.
  - For curve family PSA_ECC_FAMILY_MONTGOMERY, the scalar value of the 'public key' in little-endian order as defined by Elliptic Curves for Security [RFC7748] §6. This is a ceiling(m/8)-byte string where m is the key size in bits.
    - This is 32 bytes for Curve25519, computed as X25519(private_key, 9).
    - This is 56 bytes for Curve448, computed as X448(private_key, 5).
- For Diffie-Hellman key exchange public keys, with key types for which PSA_KEY_TYPE_IS_DH_PUBLIC_KEY is true, the format is the representation of the public key y = g^x mod p as a big-endian byte string. The length of the byte string is the length of the base prime p in bytes.

Exporting a public key object or the public part of a key pair is always permitted, regardless of the key's usage flags.

Parameter `data_size` must be appropriate for the key:

- The required output size is PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(type, bits) where type is the key type and bits is the key size in bits.

- PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key or public part of a key pair.

3.2.1.5.41.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key is neither a public key nor a key pair.
- PSA_ERROR_NOT_SUPPORTED
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `data` buffer is too small. *PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE()* or PSA_EXPORT_PUBLIC_KEY_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.42 psa_generate_key

psa_status_t **psa_generate_key**(const *psa_key_attributes_t* *attributes, *psa_key_id_t* *key)

Generate a key or key pair.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The attributes for the new key.
- **key** (*psa_key_id_t**) – On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

3.2.1.5.42.1 Description

The key is generated randomly. Its location, policy, type and size are taken from attributes.

Implementations must reject an attempt to generate a key of size 0.

The following type-specific considerations apply:

- For RSA keys (PSA_KEY_TYPE_RSA_KEY_PAIR), the public exponent is 65537. The modulus is a product of two probabilistic primes between 2^{n-1} and 2^n where n is the bit size specified in the attributes.

This function uses the attributes as follows:

- The key type is required. It cannot be an asymmetric public key.
- The key size is required. It must be a valid size for the key type.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see Permitted algorithms.
- The key usage flags define what operations are permitted with the key, see Key usage flags.
- The key lifetime and identifier are required for a persistent key.

  Note:
  This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

3.2.1.5.42.2 Return

- **PSA_SUCCESS:**
  Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
- **PSA_ERROR_ALREADY_EXISTS:**
  This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
- **PSA_ERROR_NOT_SUPPORTED:**
  The key type or key size is not supported, either by the implementation in general or in this particular persistent location.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key attributes, as a whole, are invalid.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key type is an asymmetric public key type.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key size is not a valid size for the key type.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_INSUFFICIENT_ENTROPY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE

- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_INSUFFICIENT_STORAGE
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.43 psa_generate_random

psa_status_t **psa_generate_random**(uint8_t *output, size_t output_size)

Generate random bytes.

Parameters

- **output** (uint8_t*) – Output buffer for the generated data.
- **output_size** (size_t) – Number of bytes to generate and output.

3.2.1.5.43.1 Description

Warning:
This function can fail! Callers MUST check the return status and MUST NOT use the content of the output buffer if the return status is not PSA_SUCCESS.

Note:
To generate a key, use *psa_generate_key()* instead.

3.2.1.5.43.2 Return

- PSA_SUCCESS
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INSUFFICIENT_ENTROPY
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.44 psa_get_key_algorithm

psa_algorithm_t **psa_get_key_algorithm**(const *psa_key_attributes_t* *attributes)

Retrieve the permitted algorithm policy from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.44.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.44.2 Return

typedef psa_algorithm_t

The algorithm stored in the attribute object.

3.2.1.5.45 psa_get_key_attributes

psa_status_t **psa_get_key_attributes**(*psa_key_id_t* key, *psa_key_attributes_t* *attributes)

Retrieve the attributes of a key.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to query.
- **attributes** (*psa_key_attributes_t**) – On entry, *attributes must be in a valid state. On successful return, it contains the attributes of the key. On failure, it is equivalent to a freshly-initialized attribute object.

3.2.1.5.45.1 Description

This function first resets the attribute object as with *psa_reset_key_attributes()*. It then copies the attributes of the given key into the given attribute object.

Note:

This function clears any previous content from the attribute object and therefore expects it to be in a valid state. In particular, if this function is called on a newly allocated attribute object, the attribute object must be initialized before calling this function.

Note:

This function might allocate memory or other resources. Once this function has been called on an attribute object, *psa_reset_key_attributes()* must be called to free these resources.

3.2.1.5.45.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.46 psa_get_key_bits

size_t **psa_get_key_bits**(const *psa_key_attributes_t* *attributes)

Retrieve the key size from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.46.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.46.2 Return

size_t

The key size stored in the attribute object, in bits.

3.2.1.5.47 psa_get_key_id

psa_key_id_t **psa_get_key_id**(const *psa_key_attributes_t* *attributes)

Retrieve the key identifier from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.47.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.47.2 Return

typedef psa_key_id_t

The persistent identifier stored in the attribute object. This value is unspecified if the attribute object declares the key as volatile.

3.2.1.5.48 psa_get_key_lifetime

psa_key_lifetime_t **psa_get_key_lifetime**(const *psa_key_attributes_t* *attributes)

Retrieve the lifetime from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.48.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.48.2 Return

typedef psa_key_lifetime_t

The lifetime value stored in the attribute object.

3.2.1.5.49 psa_get_key_type

psa_key_type_t **psa_get_key_type**(const *psa_key_attributes_t* *attributes)

Retrieve the key type from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.49.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.49.2 Return

typedef psa_key_type_t

The key type stored in the attribute object.

3.2.1.5.50 psa_get_key_usage_flags

psa_key_usage_t **psa_get_key_usage_flags**(const *psa_key_attributes_t* *attributes)

Retrieve the usage flags from key attributes.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The key attribute object to query.

3.2.1.5.50.1 Description

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.50.2 Return

typedef psa_key_usage_t

The usage flags stored in the attribute object.

3.2.1.5.51 psa_hash_abort

psa_status_t **psa_hash_abort**(*psa_hash_operation_t* *operation)

Abort a hash operation.

Parameters

- **operation** (*psa_hash_operation_t**) – Initialized hash operation.

3.2.1.5.51.1 Description

Warning: Not supported

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_hash_setup()* again.

This function can be called any time after the operation object has been initialized by one of the methods described in *typedef psa_hash_operation_t*.

In particular, calling *psa_hash_abort()* after the operation has been terminated by a call to *psa_hash_abort()*, *psa_hash_finish()* or *psa_hash_verify()* is safe and has no effect.

3.2.1.5.51.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.52 psa_hash_clone

psa_status_t **psa_hash_clone**(const *psa_hash_operation_t* *source_operation, *psa_hash_operation_t* *target_operation)

Clone a hash operation.

Parameters

- **source_operation** (const *psa_hash_operation_t**) – The active hash operation to clone.
- **target_operation** (*psa_hash_operation_t**) – The operation object to set up. It must be initialized but not active.

3.2.1.5.52.1 Description

Warning: Not supported

This function copies the state of an ongoing hash operation to a new operation object. In other words, this function is equivalent to calling *psa_hash_setup()* on `target_operation` with the same algorithm that `source_operation` was set up for, then *psa_hash_update()* on `target_operation` with the same input that was passed to `source_operation`. After this function returns, the two objects are independent, i.e. subsequent calls involving one of the objects do not affect the other object.

3.2.1.5.52.2 Return

- PSA_SUCCESS
- **PSA_ERROR_BAD_STATE:**
  The `source_operation` state is not valid: it must be active.
- **PSA_ERROR_BAD_STATE:**
  The `target_operation` state is not valid: it must be inactive.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_INSUFFICIENT_MEMORY

- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.53 psa_hash_compare

psa_status_t **psa_hash_compare**(*psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *hash, size_t hash_length)

Calculate the hash (digest) of a message and compare it with a reference value.

Parameters

- **alg** (*psa_algorithm_t*) – The hash algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).
- **input** (`const uint8_t*`) – Buffer containing the message to hash.
- **input_length** (`size_t`) – Size of the `input` buffer in bytes.
- **hash** (`const uint8_t*`) – Buffer containing the expected hash value.
- **hash_length** (`size_t`) – Size of the `hash` buffer in bytes.

3.2.1.5.53.1 Return

- **PSA_SUCCESS:**
  The expected hash is identical to the actual hash of the input.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The hash of the message was calculated successfully, but it differs from the expected hash.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a hash algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `input_length` or `hash_length` do not match the hash size for `alg`
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.54 psa_hash_compute

psa_status_t **psa_hash_compute**(*psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *hash, size_t hash_size, size_t *hash_length)

Calculate the hash (digest) of a message.

Parameters

- **alg** (*psa_algorithm_t*) – The hash algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).
- **input** (const uint8_t*) – Buffer containing the message to hash.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **hash** (uint8_t*) – Buffer where the hash is to be written.
- **hash_size** (size_t) – Size of the hash buffer in bytes. This must be at least PSA_HASH_LENGTH(alg).
- **hash_length** (size_t*) – On success, the number of bytes that make up the hash value. This is always PSA_HASH_LENGTH(alg).

3.2.1.5.54.1 Description

Note:

To verify the hash of a message against an expected value, use *psa_hash_compare()* instead.

3.2.1.5.54.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_NOT_SUPPORTED:**
  alg is not supported or is not a hash algorithm.
- PSA_ERROR_INVALID_ARGUMENT
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  hash_size is too small. *PSA_HASH_LENGTH()* can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_INSUFFICIENT_MEMORY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.55 psa_hash_finish

psa_status_t **psa_hash_finish**(*psa_hash_operation_t* *operation, uint8_t *hash, size_t hash_size, size_t *hash_length)

Finish the calculation of the hash of a message.

Parameters

- **operation** (*psa_hash_operation_t**) – Active hash operation.
- **hash** (uint8_t*) – Buffer where the hash is to be written.
- **hash_size** (size_t) – Size of the hash buffer in bytes. This must be at least PSA_HASH_LENGTH(alg) where alg is the algorithm that the operation performs.
- **hash_length** (size_t*) – On success, the number of bytes that make up the hash value. This is always PSA_HASH_LENGTH(alg) where alg is the hash algorithm that the operation performs.

3.2.1.5.55.1 Description

Warning: Not supported

The application must call *psa_hash_setup()* or *psa_hash_resume()* before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to *psa_hash_update()*.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_hash_abort()*.

Warning:

It is not recommended to use this function when a specific value is expected for the hash. Call *psa_hash_verify()* instead with the expected hash value.

Comparing integrity or authenticity data such as hash values with a function such as memcmp() is risky because the time taken by the comparison might leak information about the hashed data which could allow an attacker to guess a valid hash and thereby bypass security controls.

3.2.1.5.55.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the hash buffer is too small. *PSA_HASH_LENGTH()* can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE

- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.56 psa_hash_operation_init

psa_hash_operation_t **psa_hash_operation_init**(void)

Return an initial value for a hash operation object.

Parameters

- **void** – no arguments

3.2.1.5.56.1 Return

typedef psa_hash_operation_t

3.2.1.5.57 psa_hash_resume

psa_status_t **psa_hash_resume**(*psa_hash_operation_t* *operation, const uint8_t *hash_state, size_t hash_state_length)

Set up a multi-part hash operation using the hash suspend state from a previously suspended hash operation.

Parameters

- **operation** (*psa_hash_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for *typedef psa_hash_operation_t* and not yet in use.
- **hash_state** (const uint8_t*) – A buffer containing the suspended hash state which is to be resumed. This must be in the format output by *psa_hash_suspend()*, which is described in Hash suspend state format.
- **hash_state_length** (size_t) – Length of hash_state in bytes.

3.2.1.5.57.1 Description

Warning: Not supported

See *psa_hash_suspend()* for an example of how to use this function to suspend and resume a hash operation.

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

- A successful call to *psa_hash_finish()*, *psa_hash_verify()* or *psa_hash_suspend()*.
- A call to *psa_hash_abort()*.

3.2.1.5.57.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_NOT_SUPPORTED:**
  The provided hash suspend state is for an algorithm that is not supported.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `hash_state` does not correspond to a valid hash suspend state. See Hash suspend state format for the definition.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.58 psa_hash_setup

psa_status_t **psa_hash_setup**(*psa_hash_operation_t* *operation, *psa_algorithm_t* alg)

Set up a multi-part hash operation.

Parameters

- **operation** (*psa_hash_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for *typedef psa_hash_operation_t* and not yet in use.
- **alg** (*psa_algorithm_t*) – The hash algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

3.2.1.5.58.1 Description

Warning: Not supported

The sequence of operations to calculate a hash (message digest) 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 *typedef psa_hash_operation_t*, e.g. PSA_HASH_OPERATION_INIT.
3. Call *psa_hash_setup()* to specify the algorithm.
4. Call *psa_hash_update()* zero, one or more times, passing a fragment of the message each time. The hash that is calculated is the hash of the concatenation of these messages in order.

5. To calculate the hash, call *psa_hash_finish()*. To compare the hash with an expected value, call *psa_hash_verify()*. To suspend the hash operation and extract the current state, call *psa_hash_suspend()*.

If an error occurs at any step after a call to *psa_hash_setup()*, the operation will need to be reset by a call to *psa_hash_abort()*. The application can call *psa_hash_abort()* at any time after the operation has been initialized.

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

- A successful call to *psa_hash_finish()* or *psa_hash_verify()* or *psa_hash_suspend()*.
- A call to *psa_hash_abort()*.

3.2.1.5.58.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not a supported hash algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `alg` is not a hash algorithm.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.59 psa_hash_suspend

psa_status_t **psa_hash_suspend**(*psa_hash_operation_t* *operation, uint8_t *hash_state, size_t hash_state_size, size_t *hash_state_length)

Halt the hash operation and extract the intermediate state of the hash computation.

Parameters

- **operation** (*psa_hash_operation_t**) – Active hash operation.
- **hash_state** (uint8_t*) – Buffer where the hash suspend state is to be written.
- **hash_state_size** (size_t) – Size of the `hash_state` buffer in bytes.
- **hash_state_length** (size_t*) – On success, the number of bytes that make up the hash suspend state.

3.2.1.5.59.1 Description

Warning: Not supported

The application must call *psa_hash_setup()* or *psa_hash_resume()* before calling this function. This function extracts an intermediate state of the hash computation of the message formed by concatenating the inputs passed to preceding calls to *psa_hash_update()*.

This function can be used to halt a hash operation, and then resume the hash operation at a later time, or in another application, by transferring the extracted hash suspend state to a call to *psa_hash_resume()*.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_hash_abort()*.

Hash suspend and resume is not defined for the SHA3 family of hash algorithms. Hash suspend state defines the format of the output from *psa_hash_suspend()*.

Warning:

Applications must not use any of the hash suspend state as if it was a hash output. Instead, the suspend state must only be used to resume a hash operation, and *psa_hash_finish()* or *psa_hash_verify()* can then calculate or verify the final hash value.

Parameter `hash_state_size` must be appropriate for the selected algorithm:

- A sufficient output size is PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) where alg is the algorithm that was used to set up the operation.
- PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported hash algorithm.

3.2.1.5.59.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `hash_state` buffer is too small. *PSA_HASH_SUSPEND_OUTPUT_SIZE()* or PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- **PSA_ERROR_NOT_SUPPORTED:**
  The hash algorithm being computed does not support suspend and resume.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.60 psa_hash_update

psa_status_t **psa_hash_update**(*psa_hash_operation_t* *operation, const uint8_t *input, size_t input_length)

Add a message fragment to a multi-part hash operation.

Parameters

- **operation** (*psa_hash_operation_t**) – Active hash operation.
- **input** (const uint8_t*) – Buffer containing the message fragment to hash.
- **input_length** (size_t) – Size of the input buffer in bytes.

3.2.1.5.60.1 Description

Warning: Not supported

The application must call *psa_hash_setup()* or *psa_hash_resume()* before calling this function.

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

3.2.1.5.60.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.61 psa_hash_verify

psa_status_t **psa_hash_verify**(*psa_hash_operation_t* *operation, const uint8_t *hash, size_t hash_length)

Finish the calculation of the hash of a message and compare it with an expected value.

Parameters

- **operation** (*psa_hash_operation_t**) – Active hash operation.
- **hash** (const uint8_t*) – Buffer containing the expected hash value.
- **hash_length** (size_t) – Size of the hash buffer in bytes.

3.2.1.5.61.1 Description

Warning: Not supported

The application must call *psa_hash_setup()* before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to *psa_hash_update()*. It then compares the calculated hash with the expected hash passed as a parameter to this function.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_hash_abort()*.

Note:

Implementations must make the best effort to ensure that the comparison between the actual hash and the expected hash is performed in constant time.

3.2.1.5.61.2 Return

- **PSA_SUCCESS:**
  The expected hash is identical to the actual hash of the message.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The hash of the message was calculated successfully, but it differs from the expected hash.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.62 psa_import_key

psa_status_t **psa_import_key**(const *psa_key_attributes_t* *attributes, const uint8_t *data, size_t data_length, *psa_key_id_t* *key)

Import a key in binary format.

Parameters

- **attributes** (const *psa_key_attributes_t**) – The attributes for the new key.
- **data** (const uint8_t*) – Buffer containing the key data. The content of this buffer is interpreted according to the type declared in attributes. All implementations must support at least the format described in the documentation of *psa_export_key()* or *psa_export_public_key()* for the chosen type. Implementations can support other formats, but

be conservative in interpreting the key data: it is recommended that implementations reject content if it might be erroneous, for example, if it is the wrong type or is truncated.

- **data_length** (size_t) – Size of the data buffer in bytes.
- **key** (*psa_key_id_t**) – On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

3.2.1.5.62.1 Description

Warning: Import of private key may be not supported depending on the secure subsystem in use.

This function supports any output from *psa_export_key()*. Refer to the documentation of *psa_export_public_key()* for the format of public keys and to the documentation of *psa_export_key()* for the format for other key types.

The key data determines the key size. The attributes can optionally specify a key size; in this case it must match the size determined from the key data. A key size of 0 in attributes indicates that the key size is solely determined by the key data.

Implementations must reject an attempt to import a key of size 0.

This specification defines a single format for each key type. Implementations can optionally support other formats in addition to the standard format. It is recommended that implementations that support other formats ensure that the formats are clearly unambiguous, to minimize the risk that an invalid input is accidentally interpreted according to a different format.

Note:

The PSA Crypto API does not support asymmetric private key objects outside of a key pair. To import a private key, the attributes must specify the corresponding key pair type. Depending on the key type, either the import format contains the public key data or the implementation will reconstruct the public key from the private key as needed.

This function uses the attributes as follows:

- The key type is required, and determines how the data buffer is interpreted.
- The key size is always determined from the data buffer. If the key size in attributes is nonzero, it must be equal to the size determined from data.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic peration, see Permitted algorithms.
- The key usage flags define what operations are permitted with the key, see Key usage flags.
- The key lifetime and identifier are required for a persistent key.

  Note:

  This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling *psa_get_key_attributes()* with the key's identifier.

3.2.1.5.62.2 Return

- **PSA_SUCCESS:**
  Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
- **PSA_ERROR_ALREADY_EXISTS:**
  This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
- **PSA_ERROR_NOT_SUPPORTED:**
  The key type or key size is not supported, either by the implementation in general or in this particular persistent location.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key attributes, as a whole, are invalid.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key data is not correctly formatted.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The size in `attributes` is nonzero and does not match the size of the key data.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_INSUFFICIENT_STORAGE
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.63 psa_key_attributes_init

psa_key_attributes_t **psa_key_attributes_init**(void)

Return an initial value for a key attribute object.

Parameters

- **void** – no arguments

3.2.1.5.63.1 Return

typedef psa_key_attributes_t

3.2.1.5.64 psa_key_derivation_abort

psa_status_t **psa_key_derivation_abort**(*psa_key_derivation_operation_t* *operation)

Abort a key derivation operation.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The operation to abort.

3.2.1.5.64.1 Description

Warning: Not supported

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_key_derivation_setup()* again.

This function can be called at any time after the operation object has been initialized as described in *typedef psa_key_derivation_operation_t*.

In particular, it is valid to call *psa_key_derivation_abort()* twice, or to call *psa_key_derivation_abort()* on an operation that has not been set up.

3.2.1.5.64.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.65 psa_key_derivation_get_capacity

psa_status_t **psa_key_derivation_get_capacity**(const *psa_key_derivation_operation_t* *operation, size_t *capacity)

Retrieve the current capacity of a key derivation operation.

Parameters

- **operation** (const *psa_key_derivation_operation_t**) – The operation to query.

- **capacity** (size_t*) – On success, the capacity of the operation.

3.2.1.5.65.1 Description

Warning: Not supported

The capacity of a key derivation is the maximum number of bytes that it can return. Reading N bytes of output from a key derivation operation reduces its capacity by at least N. The capacity can be reduced by more than N in the following situations:

- Calling *psa_key_derivation_output_key()* can reduce the capacity by more than the key size, depending on the type of key being generated. See *psa_key_derivation_output_key()* for details of the key derivation process.
- When the *typedef psa_key_derivation_operation_t* object is operating as a deterministic random bit generator (DBRG), which reduces capacity in whole blocks, even when less than a block is read.

3.2.1.5.65.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.66 psa_key_derivation_input_bytes

psa_status_t **psa_key_derivation_input_bytes**(*psa_key_derivation_operation_t* *operation, *psa_key_derivation_step_t* step, const uint8_t *data, size_t data_length)

Provide an input for key derivation or key agreement.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to use. It must have been set up with *psa_key_derivation_setup()* and must not have produced any output yet.
- **step** (*psa_key_derivation_step_t*) – Which step the input data is for.
- **data** (const uint8_t*) – Input data to use.
- **data_length** (size_t) – Size of the data buffer in bytes.

3.2.1.5.66.1 Description

Warning: Not supported

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function passes direct inputs, which is usually correct for non-secret inputs. To pass a secret input, which is normally in a key object, call *psa_key_derivation_input_key()* instead of this function. Refer to the documentation of individual step types (PSA_KEY_DERIVATION_INPUT_xxx values of *typedef psa_key_derivation_step_t*) for more information.

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

3.2.1.5.66.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `step` is not compatible with the operation's algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `step` does not allow direct inputs.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid for this input step. This can happen if the application provides a step out of order or repeats a step that may not be repeated.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.67 psa_key_derivation_input_integer

psa_status_t **psa_key_derivation_input_integer**(*psa_key_derivation_operation_t* *operation, *psa_key_derivation_step_t* step, uint64_t value)

Provide a numeric input for key derivation or key agreement.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to use. It must have been set up with *psa_key_derivation_setup()* and must not have produced any output yet.
- **step** (*psa_key_derivation_step_t*) – Which step the input data is for.
- **value** (`uint64_t`) – The value of the numeric input.

3.2.1.5.67.1 Description

Warning: Not supported

Which inputs are required and in what order depends on the algorithm. However, when an algorithm requires a particular order, numeric inputs usually come first as they tend to be configuration parameters. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function is used for inputs which are fixed-size non-negative integers.

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

3.2.1.5.67.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The following conditions can result in this error:
  - The operation state is not valid for this input `step`. This can happen if the application provides a step out of order or repeats a step that may not be repeated.
  - The library requires initializing by a call to *psa_crypto_init()*.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The following conditions can result in this error:
  - `step` is not compatible with the operation's algorithm.
  - `step` does not allow numerical inputs.
  - `value` is not valid for `step` in the operation's algorithm.
- **PSA_ERROR_NOT_SUPPORTED:**
  The following conditions can result in this error:
  - `step` is not supported with the operation's algorithm.

- `value` is not supported for `step` in the operation's algorithm.

- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID

3.2.1.5.68 psa_key_derivation_input_key

psa_status_t **psa_key_derivation_input_key**(*psa_key_derivation_operation_t* *operation, *psa_key_derivation_step_t* step, *psa_key_id_t* key)

Provide an input for key derivation in the form of a key.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to use. It must have been set up with `psa_key_derivation_setup()` and must not have produced any output yet.
- **step** (*psa_key_derivation_step_t*) – Which step the input data is for.
- **key** (*psa_key_id_t*) – Identifier of the key. It must have an appropriate type for step and must allow the usage PSA_KEY_USAGE_DERIVE.

3.2.1.5.68.1 Description

Warning: Not supported

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function obtains input from a key object, which is usually correct for secret inputs or for non-secret personalization strings kept in the key store. To pass a non-secret parameter which is not in the key store, call `psa_key_derivation_input_bytes()` instead of this function. Refer to the documentation of individual step types (PSA_KEY_DERIVATION_INPUT_xxx values of type `typedef psa_key_derivation_step_t`) for more information.

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

3.2.1.5.68.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DERIVE flag.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `step` is not compatible with the operation's algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `step` does not allow key inputs of the given type or does not allow key inputs at all.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid for this input step. This can happen if the application provides a step out of order or repeats a step that may not be repeated.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.69 psa_key_derivation_key_agreement

psa_status_t **psa_key_derivation_key_agreement**(*psa_key_derivation_operation_t* *operation, *psa_key_derivation_step_t* step, *psa_key_id_t* private_key, const uint8_t *peer_key, size_t peer_key_length)

Perform a key agreement and use the shared secret as input to a key derivation.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to use. It must have been set up with *psa_key_derivation_setup()* with a key agreement and derivation algorithm alg (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_AGREEMENT(alg) is true and PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) is false). The operation must be ready for an input of the type given by step.
- **step** (*psa_key_derivation_step_t*) – Which step the input data is for.

- **private_key** (*psa_key_id_t*) – Identifier of the private key to use. It must allow the usage PSA_KEY_USAGE_DERIVE.
- **peer_key** (const uint8_t*) – Public key of the peer. The peer key must be in the same format that *psa_import_key()* accepts for the public key type corresponding to the type of private_key. That is, this function performs the equivalent of psa_import_key(…, peer_key, peer_key_length) where with key attributes indicating the public key type corresponding to the type of private_key. For example, for EC keys, this means that peer_key is interpreted as a point on the curve that the private key is on. The standard formats for public keys are documented in the documentation of *psa_export_public_key()*.
- **peer_key_length** (size_t) – Size of peer_key in bytes.

3.2.1.5.69.1 Description

Warning: Not supported

A key agreement algorithm takes two inputs: a private key private_key a public key peer_key. The result of this function is passed as input to a key derivation. The output of this key derivation can be extracted by reading from the resulting operation to produce keys and other cryptographic material.

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

3.2.1.5.69.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid for this key agreement step.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DERIVE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  private_@key is not compatible with alg, or peer_key is not valid for alg or not compatible with private_key.
- **PSA_ERROR_NOT_SUPPORTED:**
  alg is not supported or is not a key derivation algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  step does not allow an input resulting from a key agreement.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE

- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.70 psa_key_derivation_operation_init

psa_key_derivation_operation_t **psa_key_derivation_operation_init**(void)

Return an initial value for a key derivation operation object.

Parameters

- **void** – no arguments

3.2.1.5.70.1 Return

typedef psa_key_derivation_operation_t

3.2.1.5.71 psa_key_derivation_output_bytes

psa_status_t **psa_key_derivation_output_bytes**(*psa_key_derivation_operation_t* *operation, uint8_t *output, size_t output_length)

Read some data from a key derivation operation.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to read from.
- **output** (uint8_t*) – Buffer where the output will be written.
- **output_length** (size_t) – Number of bytes to `output`.

3.2.1.5.71.1 Description

Warning: Not supported

This function calculates output bytes from a key derivation algorithm and returns those bytes. If the key derivation's output is viewed as a stream of bytes, this function consumes the requested number of bytes from the stream and returns them to the caller. The operation's capacity decreases by the number of bytes read.

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

3.2.1.5.71.2 Return

- PSA_SUCCESS
- **PSA_ERROR_INSUFFICIENT_DATA:**
  The operation's capacity was less than `output_length` bytes. Note that in this case, no output is written to the `output` buffer. The operation's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller `output` buffer.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active and completed all required input steps.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *`psa_crypto_init()`*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.72 psa_key_derivation_output_key

psa_status_t **`psa_key_derivation_output_key`**(const *psa_key_attributes_t* *attributes,
psa_key_derivation_operation_t *operation,
psa_key_id_t *key)

Derive a key from an ongoing key derivation operation.

Parameters

- **`attributes`** (const *`psa_key_attributes_t`**) – The attributes for the new key.
- **`operation`** (*`psa_key_derivation_operation_t`**) – The key derivation operation object to read from.
- **`key`** (*`psa_key_id_t`**) – On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

3.2.1.5.72.1 Description

Warning: Not supported

This function calculates output bytes from a key derivation algorithm and uses those bytes to generate a key deterministically. The key's location, policy, type and size are taken from `attributes`.

If the key derivation's output is viewed as a stream of bytes, this function consumes the required number of bytes from the stream. The operation's capacity decreases by the number of bytes used to derive the key.

If this function returns an error status other than PSA_ERROR_INSUFFICIENT_DATA, the operation enters an error state and must be aborted by calling *psa_key_derivation_abort()*.

How much output is produced and consumed from the operation, and how the key is derived, depends on the key type. The table below describes the required key derivation procedures for standard key derivation algorithms. Implementations can use other methods for implementation-specific algorithms.

In all cases, the data that is read is discarded from the operation. The operation's capacity is decreased by the number of bytes read.

Key type	Key type details and derivation procedure
AES	PSA_KEY_TYPE_AES
ARC4	PSA_KEY_TYPE_ARC4
CAMELLIA	PSA_KEY_TYPE_CAMELLIA
ChaCha20	PSA_KEY_TYPE_CHACHA20
SM4	PSA_KEY_TYPE_SM4
Secrets for derivation	PSA_KEY_TYPE_DERIVE
HMAC	PSA_KEY_TYPE_HMAC For key types for which the key is an arbitrary sequence of bytes of a given size, this function is functionally equivalent to calling psa_key_derivation_output_bytes() and passing the resulting output to psa_import_key(). However, this function has a security benefit: if the implementation provides an isolation boundary then the key material is not exposed outside the isolation boundary. As a consequence, for these key types, this function always consumes exactly (bits/8) bytes from the operation.
DES	PSA_KEY_TYPE_DES, 64 bits. This function generates a key using the following process: 1. Draw an 8-byte string. 2. Set/clear the parity bits in each byte. 3. If the result is a forbidden weak key, discard the result and return to step 1. 4. Output the string.
2-key 3DES	PSA_KEY_TYPE_DES, 192 bits.
3-key 3DES	PSA_KEY_TYPE_DES, 128 bits. The two or three keys are generated by repeated application of the process used to generate a DES key. For example, for 3-key 3DES, if the first 8 bytes specify a weak key and the next 8 bytes do not, discard the first 8 bytes, use the next 8 bytes as the first key, and continue reading output from the operation to derive the other two keys.
Finite-field Diffie-Hellman keys	PSA_KEY_TYPE_DH_KEY_PAIR(dh_family) where dh_family designates any Diffie-Hellman family.
ECC keys on a Weierstrass elliptic curve	PSA_KEY_TYPE_ECC_KEY_PAIR(ecc_family) where ecc_family designates a Weierstrass curve family. These key types require the generation of a private key which is an integer in the range [1, N - 1], where N is the boundary of the private key domain: N is the prime p for Diffie-Hellman, or the order of the curve's base point for ECC.

For algorithms that take an input step PSA_KEY_DERIVATION_INPUT_SECRET, the input to that step must be provided with `psa_key_derivation_input_key()`. Future versions of this specification might include additional restrictions on the derived key based on the attributes and strength of the secret key.

This function uses the `attributes` as follows:

- The key type is required. It cannot be an asymmetric public key.
- The key size is required. It must be a valid size for the key type.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see Permitted algorithms.
- The key usage flags define what operations are permitted with the key, see Key usage flags.
- The key lifetime and identifier are required for a persistent key.

  Note:

  This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

3.2.1.5.72.2 Return

- **PSA_SUCCESS:**
  Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
- **PSA_ERROR_ALREADY_EXISTS:**

  This is an attempt to create a persistent key, and there is already a persistent key with the
  given identifier.
- **PSA_ERROR_INSUFFICIENT_DATA:**
  There was not enough data to create the desired key. Note that in this case, no output is written to the output buffer. The operation's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller output buffer.
- **PSA_ERROR_NOT_SUPPORTED:**
  The key type or key size is not supported, either by the implementation in general or in this particular location.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key attributes, as a whole, are invalid.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key type is an asymmetric public key type.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key size is not a valid size for the key type.
- **PSA_ERROR_NOT_PERMITTED:**
  The PSA_KEY_DERIVATION_INPUT_SECRET input was neither provided through a key nor the result of a key agreement.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active and completed all required input steps.

- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_INSUFFICIENT_STORAGE
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.73 psa_key_derivation_set_capacity

psa_status_t **psa_key_derivation_set_capacity**(*psa_key_derivation_operation_t* *operation, size_t capacity)

Set the maximum capacity of a key derivation operation.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to modify.
- **capacity** (size_t) – The new capacity of the operation. It must be less or equal to the operation's current capacity.

3.2.1.5.73.1 Description

Warning: Not supported

The capacity of a key derivation operation is the maximum number of bytes that the key derivation operation can return from this point onwards.

3.2.1.5.73.2 Return

- PSA_SUCCESS
- **PSA_ERROR_INVALID_ARGUMENT:**
  `capacity` is larger than the operation's current capacity. In this case, the operation object remains valid and its capacity remains unchanged.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED

- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.74 psa_key_derivation_setup

psa_status_t **psa_key_derivation_setup**(*psa_key_derivation_operation_t* *operation, *psa_algorithm_t* alg)

Set up a key derivation operation.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to set up. It must have been initialized but not set up yet.
- **alg** (*psa_algorithm_t*) – The key derivation algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_DERIVATION(alg) is true).

3.2.1.5.74.1 Description

Warning: Not supported

A key derivation algorithm takes some inputs and uses them to generate a byte stream in a deterministic way. This byte stream can be used to produce keys and other cryptographic material.

To derive a key:

1. Start with an initialized object of *typedef psa_key_derivation_operation_t*.
2. Call *psa_key_derivation_setup()* to select the algorithm.
3. Provide the inputs for the key derivation by calling *psa_key_derivation_input_bytes()* or *psa_key_derivation_input_key()* as appropriate. Which inputs are needed, in what order, whether keys are permitted, and what type of keys depends on the algorithm.
4. Optionally set the operation's maximum capacity with *psa_key_derivation_set_capacity()*. This can be done before, in the middle of, or after providing inputs. For some algorithms, this step is mandatory because the output depends on the maximum capacity.
5. To derive a key, call *psa_key_derivation_output_key()*. To derive a byte string for a different purpose, call *psa_key_derivation_output_bytes()*. Successive calls to these functions use successive output bytes calculated by the key derivation algorithm.
6. Clean up the key derivation operation object with *psa_key_derivation_abort()*.

If this function returns an error, the key derivation operation object is not changed.

If an error occurs at any step after a call to *psa_key_derivation_setup()*, the operation will need to be reset by a call to *psa_key_derivation_abort()*.

Implementations must reject an attempt to derive a key of size 0.

3.2.1.5.74.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `alg` is not a key derivation algorithm.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a key derivation algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.75 psa_key_derivation_verify_bytes

psa_status_t **psa_key_derivation_verify_bytes**(*psa_key_derivation_operation_t* *operation, const uint8_t *expected_output, size_t output_length)

Compare output data from a key derivation operation to an expected value.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to read from.
- **expected_output** (`const uint8_t*`) – Buffer containing the expected derivation output.
- **output_length** (`size_t`) – Length ot the expected output. This is also the number of bytes that will be read.

3.2.1.5.75.1 Description

Warning: Not supported

This function calculates output bytes from a key derivation algorithm and compares those bytes to an expected value. If the key derivation's output is viewed as a stream of bytes, this function destructively reads output_length bytes from the stream before comparing them with `expected_output`. The operation's capacity decreases by the number of bytes read.

This is functionally equivalent to the following code:

```
uint8_t tmp[output_length];
psa_key_derivation_output_bytes(operation, tmp, output_length);
if (memcmp(expected_output, tmp, output_length) != 0)
return PSA_ERROR_INVALID_SIGNATURE;
```

However, calling *psa_key_derivation_verify_bytes()* works even if the key's policy does not allow output of the bytes.

If this function returns an error status other than PSA_ERROR_INSUFFICIENT_DATA or PSA_ERROR_INVALID_SIGNATURE, the operation enters an error state and must be aborted by calling *psa_key_derivation_abort()*.

Note:

Implementations must make the best effort to ensure that the comparison between the actual key derivation output and the expected output is performed in constant time.

3.2.1.5.75.2 Return

- **PSA_SUCCESS:**
  Success. The output of the key derivation operation matches `expected_output`.
- **PSA_ERROR_BAD_STATE:**
  The following conditions can result in this error:
  - The operation state is not valid: it must be active, with all required input steps complete.
  - The library requires initializing by a call to *psa_crypto_init()*.
- **PSA_ERROR_NOT_PERMITTED:**
  One of the inputs is a key whose policy does not permit PSA_KEY_USAGE_VERIFY_DERIVATION.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The output of the key derivation operation does not match the value in `expected_output`.
- **PSA_ERROR_INSUFFICIENT_DATA:**
  The operation's capacity was less than `output_length` bytes. In this case, the operation's capacity is set to zero — subsequent calls to this function will not succeed, even with a smaller expected output length.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED

- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID

3.2.1.5.76 psa_key_derivation_verify_key

psa_status_t **psa_key_derivation_verify_key**(*psa_key_derivation_operation_t* *operation, *psa_key_id_t* expected)

Compare output data from a key derivation operation to an expected value stored in a key.

Parameters

- **operation** (*psa_key_derivation_operation_t**) – The key derivation operation object to read from.
- **expected** (*psa_key_id_t*) – A key of type PSA_KEY_TYPE_PASSWORD_HASH containing the expected output. The key must allow the usage PSA_KEY_USAGE_VERIFY_DERIVATION, and the permitted algorithm must match the operation's algorithm. The value of this key is typically computed by a previous call to `psa_key_derivation_output_key()`.

3.2.1.5.76.1 Description

Warning: Not supported

This function calculates output bytes from a key derivation algorithm and compares those bytes to an expected value, provided as key of type PSA_KEY_TYPE_PASSWORD_HASH. If the key derivation's output is viewed as a stream of bytes, this function destructively reads the number of bytes corresponding to the length of the expected key from the stream before comparing them with the key value. The operation's capacity decreases by the number of bytes read.

This is functionally equivalent to exporting the expected key and calling `psa_key_derivation_verify_bytes()` on the result, except that it works when the key cannot be exported.

If this function returns an error status other than PSA_ERROR_INSUFFICIENT_DATA or PSA_ERROR_INVALID_SIGNATURE, the operation enters an error state and must be aborted by calling `psa_key_derivation_abort()`.

Note:

Implementations must make the best effort to ensure that the comparison between the actual key derivation output and the expected output is performed in constant time.

3.2.1.5.76.2 Return

- **PSA_SUCCESS:**
  Success. The output of the key derivation operation matches the expected key value.
- **PSA_ERROR_BAD_STATE:**
  The following conditions can result in this error:
  - The operation state is not valid: it must be active, with all required input steps complete.
  - The library requires initializing by a call to *psa_crypto_init()*.
- **PSA_ERROR_INVALID_HANDLE:**
  `expected` is not a valid key identifier.
- **PSA_ERROR_NOT_PERMITTED:**
  The following conditions can result in this error:
  - The key does not have the PSA_KEY_USAGE_VERIFY_DERIVATION flag, or it does not permit the requested algorithm.
  - One of the inputs is a key whose policy does not permit PSA_KEY_USAGE_VERIFY_DERIVATION.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The output of the key derivation operation does not match the value of the expected key.
- **PSA_ERROR_INSUFFICIENT_DATA:**
  The operation's capacity was less than the length of the expected key. In this case, the operation's capacity is set to zero — subsequent calls to this function will not succeed, even with a smaller expected key length.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The key type is not PSA_KEY_TYPE_PASSWORD_HASH.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID

3.2.1.5.77 psa_mac_abort

psa_status_t **psa_mac_abort**(*psa_mac_operation_t* *operation)

Abort a MAC operation.

Parameters

- **operation** (*psa_mac_operation_t**) – Initialized MAC operation.

3.2.1.5.77.1 Description

Warning: Not supported

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_mac_sign_setup()* or *psa_mac_verify_setup()* again.

This function can be called any time after the operation object has been initialized by one of the methods described in *typedef psa_mac_operation_t*.

In particular, calling *psa_mac_abort()* after the operation has been terminated by a call to *psa_mac_abort()*, *psa_mac_sign_finish()* or *psa_mac_verify_finish()* is safe and has no effect.

3.2.1.5.77.2 Return

- PSA_SUCCESS
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.78 psa_mac_compute

psa_status_t **psa_mac_compute**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *mac, size_t mac_size, size_t *mac_length)

Calculate the message authentication code (MAC) of a message.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_SIGN_MESSAGE.
- **alg** (*psa_algorithm_t*) – The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).
- **input** (const uint8_t*) – Buffer containing the input message.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **mac** (uint8_t*) – Buffer where the MAC value is to be written.
- **mac_size** (size_t) – Size of the mac buffer in bytes.
- **mac_length** (size_t*) – On success, the number of bytes that make up the MAC value.

3.2.1.5.78.1 Description

Note:

To verify the MAC of a message against an expected value, use *psa_mac_verify()* instead. Beware that comparing integrity or authenticity data such as MAC values with a function such as `memcmp()` is risky because the time taken by the comparison might leak information about the MAC value which could allow an attacker to guess a valid MAC and thereby bypass security controls.

Parameter `mac_size` must be appropriate for the selected algorithm and key:

- The exact MAC size is PSA_MAC_LENGTH(key_type, key_bits, alg) where key_type and key_bits are attributes of the key used to compute the MAC.
- PSA_MAC_MAX_SIZE evaluates to the maximum MAC size of any supported MAC algorithm.

3.2.1.5.78.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_SIGN_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a MAC algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `mac` buffer is too small. *PSA_MAC_LENGTH()* or PSA_MAC_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_STORAGE_FAILURE:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_CORRUPT:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_INVALID:**
  The key could not be retrieved from storage.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.79 psa_mac_operation_init

psa_mac_operation_t **psa_mac_operation_init**(void)

Return an initial value for a MAC operation object.

Parameters

- **void** – no arguments

3.2.1.5.79.1 Return

typedef psa_mac_operation_t

3.2.1.5.80 psa_mac_sign_finish

psa_status_t **psa_mac_sign_finish**(*psa_mac_operation_t* *operation, uint8_t *mac, size_t mac_size, size_t *mac_length)

Finish the calculation of the MAC of a message.

Parameters

- **operation** (*psa_mac_operation_t**) – Active MAC operation.
- **mac** (uint8_t*) – Buffer where the MAC value is to be written.
- **mac_size** (size_t) – Size of the mac buffer in bytes.
- **mac_length** (size_t*) – On success, the number of bytes that make up the MAC value. This is always PSA_MAC_FINAL_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of the key and alg is the MAC algorithm that is calculated.

3.2.1.5.80.1 Description

Warning: Not supported

The application must call *psa_mac_sign_setup()* before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to *psa_mac_update()*.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_mac_abort()*.

Warning:

It is not recommended to use this function when a specific value is expected for the MAC. Call *psa_mac_verify_finish()* instead with the expected MAC value.

Comparing integrity or authenticity data such as MAC values with a function such as memcmp() is risky because the time taken by the comparison might leak information about the hashed data which could allow an attacker to guess a valid MAC and thereby bypass security controls.

Parameter mac_size must be appropriate for the selected algorithm and key:

- The exact MAC size is PSA_MAC_LENGTH(key_type, key_bits, alg) where key_type and key_bits are attributes of the key, and alg is the algorithm used to compute the MAC.
- PSA_MAC_MAX_SIZE evaluates to the maximum MAC size of any supported MAC algorithm.

3.2.1.5.80.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be an active mac sign operation.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `mac` buffer is too small. *PSA_MAC_LENGTH()* or PSA_MAC_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.81 psa_mac_sign_setup

psa_status_t **psa_mac_sign_setup**(*psa_mac_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set up a multi-part MAC calculation operation.

Parameters

- **operation** (*psa_mac_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for psa_mac_operation_t and not yet in use.
- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_SIGN_MESSAGE.
- **alg** (*psa_algorithm_t*) – The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).

3.2.1.5.81.1 Description

Warning: Not supported

This function sets up the calculation of the message authentication code (MAC) of a byte string. To verify the MAC of a message against an expected value, use *psa_mac_verify_setup()* instead.

The sequence of operations to calculate a MAC 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 *typedef psa_mac_operation_t*, e.g. PSA_MAC_OPERATION_INIT.
3. Call *psa_mac_sign_setup()* to specify the algorithm and key.
4. Call *psa_mac_update()* zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call *psa_mac_sign_finish()* to finish calculating the MAC value and retrieve it.

If an error occurs at any step after a call to *psa_mac_sign_setup()*, the operation will need to be reset by a call to *psa_mac_abort()*. The application can call *psa_mac_abort()* at any time after the operation has been initialized.

After a successful call to *psa_mac_sign_setup()*, the application must eventually terminate the operation through one of the following methods:

- A successful call to *psa_mac_sign_finish()*.
- A call to *psa_mac_abort()*.

3.2.1.5.81.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_SIGN_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a MAC algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_STORAGE_FAILURE:**
  The key could not be retrieved from storage.

- **PSA_ERROR_DATA_CORRUPT:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_INVALID:**
  The key could not be retrieved from storage.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.82 psa_mac_update

psa_status_t **psa_mac_update**(*psa_mac_operation_t* *operation, const uint8_t *input, size_t input_length)

Add a message fragment to a multi-part MAC operation.

Parameters

- **operation** (*psa_mac_operation_t**) – Active MAC operation.
- **input** (`const uint8_t*`) – Buffer containing the message fragment to add to the MAC calculation.
- **input_length** (`size_t`) – Size of the `input` buffer in bytes.

3.2.1.5.82.1 Description

Warning: Not supported

The application must call *psa_mac_sign_setup()* or *psa_mac_verify_setup()* before calling this function.

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

3.2.1.5.82.2 Return

- **PSA_SUCCESS:**
  Success.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be active.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE

- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.83 psa_mac_verify

psa_status_t **psa_mac_verify**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *mac, size_t mac_length)

Calculate the MAC of a message and compare it with a reference value.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
- **alg** (*psa_algorithm_t*) – The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).
- **input** (`const uint8_t*`) – Buffer containing the input message.
- **input_length** (`size_t`) – Size of the `input` buffer in bytes.
- **mac** (`const uint8_t*`) – Buffer containing the expected MAC value.
- **mac_length** (`size_t`) – Size of the `mac` buffer in bytes.

3.2.1.5.83.1 Return

- **PSA_SUCCESS:**
  The expected MAC is identical to the actual MAC of the input.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The MAC of the message was calculated successfully, but it differs from the expected value.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a MAC algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED

- **PSA_ERROR_STORAGE_FAILURE:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_CORRUPT:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_INVALID:**
  The key could not be retrieved from storage.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.84 psa_mac_verify_finish

psa_status_t **psa_mac_verify_finish**(*psa_mac_operation_t* *operation, const uint8_t *mac, size_t mac_length)

Finish the calculation of the MAC of a message and compare it with an expected value.

Parameters

- **operation** (*psa_mac_operation_t**) – Active MAC operation.
- **mac** (const uint8_t*) – Buffer containing the expected MAC value.
- **mac_length** (size_t) – Size of the mac buffer in bytes.

3.2.1.5.84.1 Description

Warning: Not supported

The application must call *psa_mac_verify_setup()* before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to *psa_mac_update()*. It then compares the calculated MAC with the expected MAC passed as a parameter to this function.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling *psa_mac_abort()*.

Note:
Implementations must make the best effort to ensure that the comparison between the actual MAC and the expected MAC is performed in constant time.

3.2.1.5.84.2 Return

- **PSA_SUCCESS:**
  The expected MAC is identical to the actual MAC of the message.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The MAC of the message was calculated successfully, but it differs from the expected MAC.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be an active mac verify operation.
- PSA_ERROR_INSUFFICIENT_MEMORY

- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.85 psa_mac_verify_setup

psa_status_t **psa_mac_verify_setup**(*psa_mac_operation_t* *operation, *psa_key_id_t* key, *psa_algorithm_t* alg)

Set up a multi-part MAC verification operation.

Parameters

- **operation** (*psa_mac_operation_t**) – The operation object to set up. It must have been initialized as per the documentation for `typedef psa_mac_operation_t` and not yet in use.
- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
- **alg** (*psa_algorithm_t*) – The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).

3.2.1.5.85.1 Description

Warning: Not supported

This function sets up the verification of the message authentication code (MAC) of a byte string against an expected value.

The sequence of operations to verify a MAC 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 `typedef psa_mac_operation_t`, e.g. PSA_MAC_OPERATION_INIT.
3. Call `psa_mac_verify_setup()` to specify the algorithm and key.
4. Call `psa_mac_update()` zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call `psa_mac_verify_finish()` to finish calculating the actual MAC of the message and verify it against the expected value.

If an error occurs at any step after a call to *psa_mac_verify_setup()*, the operation will need to be reset by a call to *psa_mac_abort()*. The application can call *psa_mac_abort()* at any time after the operation has been initialized.

After a successful call to *psa_mac_verify_setup()*, the application must eventually terminate the operation through one of the following methods:

- A successful call to *psa_mac_verify_finish()*.
- A call to *psa_mac_abort()*.

3.2.1.5.85.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `key` is not compatible with `alg`.
- **PSA_ERROR_NOT_SUPPORTED:**
  `alg` is not supported or is not a MAC algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_STORAGE_FAILURE:**
  The key could not be retrieved from storage
- **PSA_ERROR_DATA_CORRUPT:**
  The key could not be retrieved from storage.
- **PSA_ERROR_DATA_INVALID:**
  The key could not be retrieved from storage.
- **PSA_ERROR_BAD_STATE:**
  The operation state is not valid: it must be inactive.
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.86 psa_purge_key

psa_status_t **psa_purge_key**(*psa_key_id_t* key)

Remove non-essential copies of key material from memory.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to purge.

3.2.1.5.86.1 Description

Warning: Not supported

For keys that have been created with the PSA_KEY_USAGE_CACHE usage flag, an implementation is permitted to make additional copies of the key material that are not in storage and not for the purpose of ongoing operations.

This function will remove these extra copies of the key material from memory.

This function is not required to remove key material from memory in any of the following situations:

- The key is currently in use in a cryptographic operation.
- The key is volatile.

3.2.1.5.86.2 Return

- **PSA_SUCCESS:**
  The key material will have been removed from memory if it is not currently required.
- PSA_ERROR_INVALID_HANDLE
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_CORRUPTION_DETECTED
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.87 psa_raw_key_agreement

psa_status_t **psa_raw_key_agreement**(*psa_algorithm_t* alg, *psa_key_id_t* private_key, const uint8_t *peer_key, size_t peer_key_length, uint8_t *output, size_t output_size, size_t *output_length)

Perform a key agreement and return the raw shared secret.

Parameters

- **alg** (*psa_algorithm_t*) – The key agreement algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) is true).
- **private_key** (*psa_key_id_t*) – Identifier of the private key to use. It must allow the usage PSA_KEY_USAGE_DERIVE.
- **peer_key** (const uint8_t*) – Public key of the peer. It must be in the same format that *psa_import_key()* accepts. The standard formats for public keys are documented in the documentation of *psa_export_public_key()*.
- **peer_key_length** (size_t) – Size of peer_key in bytes.
- **output** (uint8_t*) – Buffer where the raw shared secret is to be written.
- **output_size** (size_t) – Size of the output buffer in bytes.
- **output_length** (size_t*) – On success, the number of bytes that make up the returned output.

3.2.1.5.87.1 Description

Warning: Not supported

Warning:

The raw result of a key agreement algorithm such as finite-field Diffie-Hellman or elliptic curve Diffie-Hellman has biases, and is not suitable for use as key material. Instead it is recommended that the result is used as input to a key derivation algorithm. To chain a key agreement with a key derivation, use *psa_key_derivation_key_agreement()* and other functions from the key derivation interface.

Parameter output_size must be appropriate for the keys:

- The required output size is PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(type, bits) where type is the type of private_key and bits is the bit-size of either private_key or the peer_key.
- PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported raw key agreement algorithm.

3.2.1.5.87.2 Return

- **PSA_SUCCESS:**
  Success.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_DERIVE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_ARGUMENT:**
  alg is not a key agreement algorithm
- **PSA_ERROR_INVALID_ARGUMENT:**
  private_@key is not compatible with alg, or peer_key is not valid for alg or not compatible with private_key.

- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the output buffer is too small. *PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE()* or PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
- **PSA_ERROR_NOT_SUPPORTED:**
  alg is not a supported key agreement algorithm.
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.88 psa_reset_key_attributes

void **psa_reset_key_attributes**(*psa_key_attributes_t* *attributes)

Reset a key attribute object to a freshly initialized state.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to reset.

3.2.1.5.88.1 Description

The attribute object must be initialized as described in the documentation of the type *typedef psa_key_attributes_t* before calling this function. Once the object has been initialized, this function can be called at any time.

This function frees any auxiliary resources that the object might contain.

3.2.1.5.88.2 Return

void

3.2.1.5.89 psa_set_key_algorithm

void **psa_set_key_algorithm**(*psa_key_attributes_t* *attributes, *psa_algorithm_t* alg)

Declare the permitted algorithm policy for a key.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **alg** (*psa_algorithm_t*) – The permitted algorithm to write.

3.2.1.5.89.1 Description

The permitted algorithm policy of a key encodes which algorithm or algorithms are permitted to be used with this key.

This function overwrites any permitted algorithm policy previously set in `attributes`.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.89.2 Return

void

3.2.1.5.90 psa_set_key_bits

void **psa_set_key_bits**(*psa_key_attributes_t* *attributes, size_t bits)

Declare the size of a key.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **bits** (`size_t`) – The key size in bits. If this is 0, the key size in `attributes` becomes unspecified. Keys of size 0 are not supported.

3.2.1.5.90.1 Description

This function overwrites any key size previously set in `attributes`.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.90.2 Return

void

3.2.1.5.91 psa_set_key_id

void **psa_set_key_id**(*psa_key_attributes_t* *attributes, *psa_key_id_t* id)

Declare a key as persistent and set its key identifier.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **id** (*psa_key_id_t*) – The persistent identifier for the key.

3.2.1.5.91.1 Description

The application must choose a value for `id` between PSA_KEY_ID_USER_MIN and PSA_KEY_ID_USER_MAX.

If the attribute object currently declares the key as volatile, which is the default lifetime of an attribute object, this function sets the lifetime attribute to PSA_KEY_LIFETIME_PERSISTENT.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as *psa_import_key()*, *psa_generate_key()*, *psa_key_derivation_output_key()* or *psa_copy_key()*.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.91.2 Return

void

3.2.1.5.92 psa_set_key_lifetime

void **psa_set_key_lifetime**(*psa_key_attributes_t* *attributes, *psa_key_lifetime_t* lifetime)

Set the location of a persistent key.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **lifetime** (*psa_key_lifetime_t*) – The lifetime for the key. If this is PSA_KEY_LIFETIME_VOLATILE, the key will be volatile, and the key identifier attribute is reset to PSA_KEY_ID_NULL.

3.2.1.5.92.1 Description

To make a key persistent, give it a persistent key identifier by using *psa_set_key_id()*. By default, a key that has a persistent identifier is stored in the default storage area identifier by PSA_KEY_LIFETIME_PERSISTENT. Call this function to choose a storage area, or to explicitly declare the key as volatile.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as *psa_import_key()*, *psa_generate_key()*, *psa_key_derivation_output_key()* or *psa_copy_key()*.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.92.2 Return

void

3.2.1.5.93 psa_set_key_type

void **psa_set_key_type**(*psa_key_attributes_t* *attributes, *psa_key_type_t* type)

Declare the type of a key.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **type** (*psa_key_type_t*) – The key type to write. If this is PSA_KEY_TYPE_NONE, the key type in `attributes` becomes unspecified.

3.2.1.5.93.1 Description

This function overwrites any key type previously set in `attributes`.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.93.2 Return

void

3.2.1.5.94 psa_set_key_usage_flags

void **psa_set_key_usage_flags**(*psa_key_attributes_t* *attributes, *psa_key_usage_t* usage_flags)

Declare usage flags for a key.

Parameters

- **attributes** (*psa_key_attributes_t**) – The attribute object to write to.
- **usage_flags** (*psa_key_usage_t*) – psa_set_key_usage_flags The usage flags to write.

3.2.1.5.94.1 Description

Usage flags are part of a key's policy. They encode what kind of operations are permitted on the key. For more details, see Key policies.

This function overwrites any usage flags previously set in `attributes`.

Implementation note:

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as static or inline, instead of using the default external linkage.

- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

3.2.1.5.94.2 Return

void

3.2.1.5.95 psa_sign_hash

psa_status_t **psa_sign_hash**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *hash, size_t hash_length, uint8_t *signature, size_t signature_size, size_t *signature_length)

Sign an already-calculated hash with a private key.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must be an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_SIGN_HASH.
- **alg** (*psa_algorithm_t*) – An asymmetric signature algorithm that separates the hash and sign operations (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_HASH(alg) is true), that is compatible with the type of key.
- **hash** (`const uint8_t*`) – The input to sign. This is usually the hash of a message. See the detailed description of this function and the description of individual signature algorithms for a detailed description of acceptable inputs.
- **hash_length** (`size_t`) – Size of the hash buffer in bytes.
- **signature** (`uint8_t*`) – Buffer where the signature is to be written.
- **signature_size** (`size_t`) – Size of the `signature` buffer in bytes.
- **signature_length** (`size_t*`) – On success, the number of bytes that make up the returned signature value.

3.2.1.5.95.1 Description

With most signature mechanisms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to sign. The hash algorithm is encoded in the signature algorithm.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature algorithm: PSA_ALG_RSA_PKCS1V15_SIGN_RAW.

Note:

To perform a hash-and-sign algorithm, the hash must be calculated before passing it to this function. This can be done by calling *psa_hash_compute()* or with a multi-part hash operation. Alternatively, to hash and sign a message in a single call, use *psa_sign_message()*.

Parameter `signature_size` must be appropriate for the selected algorithm and key:

- The required signature size is PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of `key`.
- PSA_SIGNATURE_MAX_SIZE evaluates to the maximum signature size of any supported signature algorithm.

3.2.1.5.95.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_SIGN_HASH flag, or it does not permit the requested algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the `signature` buffer is too small. *PSA_SIGN_OUTPUT_SIZE()* or PSA_SIGNATURE_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_ENTROPY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.96 psa_sign_message

psa_status_t **psa_sign_message**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, uint8_t *signature, size_t signature_size, size_t *signature_length)

Sign a message with a private key. For hash-and-sign algorithms, this includes the hashing step.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must be an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_SIGN_MESSAGE.

- **alg** (*psa_algorithm_t*) – An asymmetric signature algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_MESSAGE(alg) is true), that is compatible with the type of key.
- **input** (const uint8_t*) – The input message to sign.
- **input_length** (size_t) – Size of the input buffer in bytes.
- **signature** (uint8_t*) – Buffer where the signature is to be written.
- **signature_size** (size_t) – Size of the signature buffer in bytes.
- **signature_length** (size_t*) – On success, the number of bytes that make up the returned signature value.

3.2.1.5.96.1 Description

Note:

To perform a multi-part hash-and-sign signature algorithm, first use a multi-part hash operation and then pass the resulting hash to *psa_sign_hash()*. PSA_ALG_GET_HASH(alg) can be used to determine the hash algorithm to use.

Parameter signature_size must be appropriate for the selected algorithm and key:

- The required signature size is PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of key.
- PSA_SIGNATURE_MAX_SIZE evaluates to the maximum signature size of any supported signature algorithm.

3.2.1.5.96.2 Return

- PSA_SUCCESS
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_SIGN_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  The size of the signature buffer is too small. *PSA_SIGN_OUTPUT_SIZE()* or PSA_SIGNATURE_MAX_SIZE can be used to determine the required buffer size.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT

- PSA_ERROR_DATA_INVALID
- PSA_ERROR_INSUFFICIENT_ENTROPY
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.97 psa_verify_hash

psa_status_t **psa_verify_hash**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *hash, size_t hash_length, const uint8_t *signature, size_t signature_length)

Verify the signature of a hash or short message using a public key.

Parameters

- **key** (*psa_key_id_t*) – Identifier of the key to use for the operation. It must be a public key or an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_VERIFY_HASH.
- **alg** (*psa_algorithm_t*) – An asymmetric signature algorithm that separates the hash and sign operations (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_HASH(alg) is true), that is compatible with the type of key.
- **hash** (`const uint8_t*`) – The input whose signature is to be verified. This is usually the hash of a message. See the detailed description of this function and the description of individual signature algorithms for a detailed description of acceptable inputs.
- **hash_length** (`size_t`) – Size of the `hash` buffer in bytes.
- **signature** (`const uint8_t*`) – Buffer containing the signature to verify.
- **signature_length** (`size_t`) – Size of the signature buffer in bytes.

3.2.1.5.97.1 Description

With most signature mechanisms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to sign. The hash algorithm is encoded in the signature algorithm.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature algorithm: PSA_ALG_RSA_PKCS1V15_SIGN_RAW.

Note:

To perform a hash-and-sign verification algorithm, the hash must be calculated before passing it to this function. This can be done by calling *psa_hash_compute()* or with a multi-part hash operation. Alternatively, to hash and verify a message signature in a single call, use *psa_verify_message()*.

3.2.1.5.97.2 Return

- **PSA_SUCCESS:**
  The signature is valid.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_VERIFY_HASH flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The calculation was performed successfully, but the passed signature is not a valid signature.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.1.5.98 psa_verify_message

psa_status_t **`psa_verify_message`**(*psa_key_id_t* key, *psa_algorithm_t* alg, const uint8_t *input, size_t input_length, const uint8_t *signature, size_t signature_length)

Verify the signature of a message with a public key, using a hash-and-sign verification algorithm.

Parameters

- **`key`** (`psa_key_id_t`) – Identifier of the key to use for the operation. It must be a public key or an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
- **`alg`** (`psa_algorithm_t`) – An asymmetric signature algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_MESSAGE(alg) is true), that is compatible with the type of key.
- **`input`** (`const uint8_t*`) – The message whose signature is to be verified.
- **`input_length`** (`size_t`) – Size of the `input` buffer in bytes.
- **`signature`** (`const uint8_t*`) – Buffer containing the signature to verify.
- **`signature_length`** (`size_t`) – Size of the `signature` buffer in bytes.

3.2.1.5.98.1 Description

Note:

To perform a multi-part hash-and-sign signature verification algorithm, first use a multi-part hash operation to hash the message and then pass the resulting hash to *psa_verify_hash()*. PSA_ALG_GET_HASH(alg) can be used to determine the hash algorithm to use.

3.2.1.5.98.2 Return

- **PSA_SUCCESS:**
  The signature is valid.
- PSA_ERROR_INVALID_HANDLE
- **PSA_ERROR_NOT_PERMITTED:**
  The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The calculation was performed successfully, but the passed signature is not a valid signature.
- PSA_ERROR_NOT_SUPPORTED
- PSA_ERROR_INVALID_ARGUMENT
- PSA_ERROR_INSUFFICIENT_MEMORY
- PSA_ERROR_COMMUNICATION_FAILURE
- PSA_ERROR_HARDWARE_FAILURE
- PSA_ERROR_CORRUPTION_DETECTED
- PSA_ERROR_STORAGE_FAILURE
- PSA_ERROR_DATA_CORRUPT
- PSA_ERROR_DATA_INVALID
- **PSA_ERROR_BAD_STATE:**
  The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

3.2.2 Initial Attestation APIs

3.2.2.1 Introduction

The PSA Attestation API is a standard interface provided by the PSA Root of Trust. The definition of the PSA Root of Trust is described in the PSA Security Model (PSA-SM - https://www.arm.com/architecture/security-features).

The API can be used either to directly sign data or as a way to bootstrap trust in other attestation schemes. PSA provides a framework and the minimal generic security features allowing OEM and service providers to integrate various attestation schemes on top of the PSA Root of Trust.

3.2.2.2 Reference

Documentation:
PSA Attestation API v1.0.2

Link:
https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0085-PSA_Attestation_API-1.0.2.pdf

3.2.2.3 psa_initial_attest_get_token

psa_status_t **psa_initial_attest_get_token**(const uint8_t *auth_challenge, size_t challenge_size, uint8_t *token_buf, size_t token_buf_size, size_t *token_size)

Retrieve the Initial Attestation Token.

Parameters

- **auth_challenge** (const uint8_t*) – Buffer with a challenge object. The challenge object is data provided by the caller. For example, it may be a cryptographic nonce or a hash of data (such as an external object record). If a hash of data is provided then it is the caller's responsibility to ensure that the data is protected against replay attacks (for example, by including a cryptographic nonce within the data).
- **challenge_size** (size_t) – Size of the buffer auth_challenge in bytes. The size must always be a supported challenge size. Supported challenge sizes are defined by the PSA_INITIAL_ATTEST_CHALLENGE_SIZE_xxx constant.
- **token_buf** (uint8_t*) – Output buffer where the attestation token is to be written.
- **token_buf_size** (size_t) – Size of token_buf. The expected size can be determined by using *psa_initial_attest_get_token_size()*.
- **token_size** (size_t*) – Output variable for the actual token size.

3.2.2.3.1 Description

Warning: Not supported

Retrieves the Initial Attestation Token. A challenge can be passed as an input to mitigate replay attacks.

3.2.2.3.2 Return

- **PSA_SUCCESS:**
  Action was performed successfully.
- **PSA_ERROR_SERVICE_FAILURE:**
  The implementation failed to fully initialize.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  `token_buf` is too small for the attestation token.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The challenge size is not supported.
- **PSA_ERROR_GENERIC_ERROR:**
  An unspecified internal error has occurred.

3.2.2.4 psa_initial_attest_get_token_size

psa_status_t **`psa_initial_attest_get_token_size`**(size_t challenge_size, size_t *token_size)

Calculate the size of an Initial Attestation Token.

Parameters

- **`challenge_size`** (`size_t`) – Size of a challenge object in bytes. This must be a supported challenge size as defined by the PSA_INITIAL_ATTEST_CHALLENGE_SIZE_xxx constant.
- **`token_size`** (`size_t*`) – Output variable for the token size.

3.2.2.4.1 Description

Warning: Not supported

Retrieve the exact size of the Initial Attestation Token in bytes, given a specific challenge size.

3.2.2.4.2 Return

- **PSA_SUCCESS:**
  Action was performed successfully.
- **PSA_ERROR_SERVICE_FAILURE:**
  The implementation failed to fully initialize.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The challenge size is not supported.
- **PSA_ERROR_GENERIC_ERROR:**
  An unspecified internal error has occurred.

3.2.2.5 psa_attest_key

psa_status_t **psa_attest_key**(*psa_key_id_t* key, const uint8_t *auth_challenge, size_t *challenge_size, uint8_t *cert_buf, size_t cert_buf_size, size_t *cert_size)

Retrieve a Key Attestation.

Parameters

- **key** (*psa_key_id_t*) – Key identifier.
- **auth_challenge** (const uint8_t*) – Buffer with a challenge object. The challenge object is data provided by the caller. For example, it may be a cryptographic nonce or a hash of data (such as an external object record). If a hash of data is provided then it is the caller's responsibility to ensure that the data is protected against replay attacks (for example, by including a cryptographic nonce within the data).
- **challenge_size** (size_t*) – Size of a challenge object in bytes. This must be a supported challenge size as defined by the PSA_INITIAL_ATTEST_CHALLENGE_SIZE_xxx constant.
- **cert_buf** (uint8_t*) – Output variable for the Key Attestation certificate.
- **cert_buf_size** (size_t) – Maximum size of the Key Attestation certificate.
- **cert_size** (size_t*) – Output variable for the actual Key Attestation certificate size.

3.2.2.5.1 Description

Warning: Not supported

Retrieves the Key Attestation certificate. A challenge can be passed as an input to mitigate replay attacks.

3.2.2.5.2 Return

- **PSA_SUCCESS:**
  Action was performed successfully.
- **PSA_ERROR_SERVICE_FAILURE:**
  The implementation failed to fully initialize.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The challenge size is not supported.
- **PSA_ERROR_GENERIC_ERROR:**
  An unspecified internal error has occurred.

3.2.2.6 psa_attest_key_get_size

psa_status_t **psa_attest_key_get_size**(*psa_key_id_t* key, size_t challenge_size, size_t *cert_size)

Calculate the size of a Key Attestation certificate.

Parameters

- **key** (*psa_key_id_t*) – Key identifier.
- **challenge_size** (size_t) – Size of a challenge object in bytes. This must be a supported challenge size as defined by the PSA_INITIAL_ATTEST_CHALLENGE_SIZE_xxx constant.
- **cert_size** (size_t*) – Output variable for the certificate size.

3.2.2.6.1 Description

Warning: Not supported

Retrieve the exact size of the Key Attestation certificate in bytes, given a specific challenge size.

3.2.2.6.2 Return

- **PSA_SUCCESS:**
  Action was performed successfully.
- **PSA_ERROR_SERVICE_FAILURE:**
  The implementation failed to fully initialize.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The challenge size is not supported.
- **PSA_ERROR_GENERIC_ERROR:**
  An unspecified internal error has occurred.

3.2.3 Storage APIs

3.2.3.1 Storage common APIs

3.2.3.1.1 Introduction

This file defines common definitions for PSA storage.

3.2.3.1.2 Reference

Documentation:
PSA Storage API v1.0.0 section 5.1 General Definitions

Link:
https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0087-PSA_Storage_API-1.0.0.pdf

3.2.3.1.3 typedef psa_storage_create_flags_t

type **psa_storage_create_flags_t**
Storage create flags

3.2.3.1.3.1 Description

Flags used when creating a data entry.

3.2.3.1.3.2 Values

- **PSA_STORAGE_FLAG_NONE:**
  No flags to pass.
- **PSA_STORAGE_FLAG_WRITE_ONCE:**
  The data associated with the uid will not be able to be modified or deleted. Intended to be used to set bits in *typedef psa_storage_create_flags_t*.
- **PSA_STORAGE_FLAG_NO_CONFIDENTIALITY:**
  The data associated with the uid is public and therefore does not require confidentiality. It therefore only needs to be integrity protected.
- **PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION:**
  The data associated with the uid does not require replay protection. This may permit faster storage - but it permits an attacker with physical access to revert to an earlier version of the data.

3.2.3.1.4 typedef psa_storage_uid_t

type **psa_storage_uid_t**
Storage uid

3.2.3.1.4.1 Description

A type for uid used for identifying data.

3.2.3.1.5 struct psa_storage_info_t

struct **psa_storage_info_t**

Storage info

3.2.3.1.5.1 Definition

```
struct psa_storage_info_t {
    size_t capacity;
    size_t size;
    psa_storage_create_flags_t flags;
}
```

3.2.3.1.5.2 Members

capacity

The allocated capacity of the storage associated with a uid.

size

The size of the data associated with a uid.

flags

The flags set when the uid was created.

3.2.3.2 Internal trusted storage APIs

3.2.3.2.1 Introduction

The PSA Internal Trusted Storage API (PITS API) is a more specialized API. Uses of this API will be less common. It is intended to be used for assets that must be placed inside internal flash. Some examples of assets that require this are replay protection values for external storage and keys for use by components of the PSA Root of Trust.

3.2.3.2.2 Reference

Documentation:

PSA Storage API v1.0.0 section 5.2 Internal Trusted Storage API

Link:

https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0087-PSA_Storage_API-1.0.0.pdf

3.2.3.2.3 PSA_ITS_API_VERSION_MAJOR

The major version number of the PSA ITS API.

It will be incremented on significant updates that may include breaking changes.

3.2.3.2.4 PSA_ITS_API_VERSION_MINOR

The minor version number of the PSA ITS API.

It will be incremented in small updates that are unlikely to include breaking changes.

3.2.3.2.5 psa_its_set

psa_status_t **psa_its_set**(*psa_storage_uid_t* uid, size_t data_length, const void *p_data, *psa_storage_create_flags_t* create_flags)

Create a new, or modify an existing, uid/value pair.

Parameters

- **uid** (*psa_storage_uid_t*) – The identifier for the data.
- **data_length** (size_t) – The size in bytes of the data in p_data.
- **p_data** (const void*) – A buffer containing the data.
- **create_flags** (*psa_storage_create_flags_t*) – The flags that the data will be stored with.

3.2.3.2.5.1 Description

Stores data in the internal storage.

3.2.3.2.5.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_NOT_PERMITTED:**
  The operation failed because the provided uid value was already created with PSA_STORAGE_FLAG_WRITE_ONCE.
- **PSA_ERROR_NOT_SUPPORTED:**
  The operation failed because one or more of the flags provided in create_flags is not supported or is not valid.
- **PSA_ERROR_INSUFFICIENT_STORAGE:**
  The operation failed because there was insufficient space on the storage medium.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).

- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one of the provided pointers (e.g. `p_data`) is invalid, for example is NULL or references memory the caller cannot access.

3.2.3.2.6 psa_its_get

psa_status_t **psa_its_get**(*psa_storage_uid_t* uid, size_t data_offset, size_t data_size, void *p_data, size_t *p_data_length)

Retrieve data associated with a provided UID.

Parameters

- **uid** (*psa_storage_uid_t*) – The uid value.
- **data_offset** (`size_t`) – The starting offset of the data requested.
- **data_size** (`size_t`) – The amount of data requested.
- **p_data** (`void*`) – On success, the buffer where the data will be placed.
- **p_data_length** (`size_t*`) – On success, this will contain size of the data placed in `p_data`.

3.2.3.2.6.1 Description

Retrieves up to `data_size` bytes of the data associated with `uid`, starting at `data_offset` bytes from the beginning of the data. Upon successful completion, the data will be placed in the `p_data` buffer, which must be at least `data_size` bytes in size. The length of the data returned will be in `p_data_length`. If `data_size` is 0, the contents of `p_data_length` will be set to zero.

3.2.3.2.6.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided `uid` value was not found in the storage.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one of the provided arguments (e.g. `p_data`, `p_data_length`) is invalid, for example is NULL or references memory the caller cannot access. In addition, this can also happen if `data_offset` is larger than the size of the data associated with `uid`.

3.2.3.2.7 psa_its_get_info

psa_status_t **psa_its_get_info**(*psa_storage_uid_t* uid, struct *psa_storage_info_t* *p_info)

Retrieve the metadata about the provided `uid`.

Parameters

- **uid** (*psa_storage_uid_t*) – The uid value.
- **p_info** (struct *psa_storage_info_t**) – A pointer to the *struct psa_storage_info_t* that will be populated with the metadata.

3.2.3.2.7.1 Description

Retrieves the metadata stored for a given `uid` as a *struct psa_storage_info_t*.

3.2.3.2.7.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided `uid` value was not found in the storage.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one of the provided pointers (e.g. `p_info`) is invalid, for example is NULL or references memory the caller cannot access.

3.2.3.2.8 psa_its_remove

psa_status_t **psa_its_remove**(*psa_storage_uid_t* uid)

Remove the provided key and its associated data from the storage.

Parameters

- **uid** (*psa_storage_uid_t*) – The uid value.

3.2.3.2.8.1 Description

Deletes the data from internal storage.

3.2.3.2.8.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on).
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided key value was not found in the storage.
- **PSA_ERROR_NOT_PERMITTED:**
  The operation failed because the provided key value was created with PSA_STORAGE_FLAG_WRITE_ONCE.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).

3.2.3.3 Protected storage APIs

3.2.3.3.1 Introduction

The PSA Protected Storage API (PS API) is the general-purpose API that most developers should use. It is intended to be used to protect storage media that are external to the MCU package.

3.2.3.3.2 Reference

Documentation:
PSA Storage API v1.0.0 section 5.3 Protected Storage API

Link:
https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0087-PSA_Storage_API-1.0.0.pdf

3.2.3.3.3 PSA_PS_API_VERSION_MAJOR

The major version number of the PSA PS API.

It will be incremented on significant updates that may include breaking changes.

3.2.3.3.4 PSA_PS_API_VERSION_MINOR

The minor version number of the PSA PS API.

It will be incremented in small updates that are unlikely to include breaking changes.

3.2.3.3.5 psa_ps_set

psa_status_t **psa_ps_set**(*psa_storage_uid_t* uid, size_t data_length, const void *p_data, *psa_storage_create_flags_t* create_flags)

Create a new or modify an existing key/value pair.

Parameters

- **uid** (*psa_storage_uid_t*) – The identifier for the data.
- **data_length** (size_t) – The size in bytes of the data in p_data.
- **p_data** (const void*) – A buffer containing the data.
- **create_flags** (*psa_storage_create_flags_t*) – The flags indicating the properties of the data.

3.2.3.3.5.1 Description

Warning: Not supported

The newly created asset has a capacity and size that are equal to data_length.

3.2.3.3.5.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_NOT_PERMITTED:**
  The operation failed because the provided uid value was already created with PSA_STORAGE_FLAG_WRITE_ONCE.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one or more of the given arguments were invalid.
- **PSA_ERROR_NOT_SUPPORTED:**
  The operation failed because one or more of the flags provided in create_flags is not supported or is not valid.
- **PSA_ERROR_INSUFFICIENT_STORAGE:**
  The operation failed because there was insufficient space on the storage medium.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_GENERIC_ERROR:**
  The operation failed because of an unspecified internal failure.

3.2.3.3.6 psa_ps_get

psa_status_t **psa_ps_get**(*psa_storage_uid_t* uid, size_t data_offset, size_t data_size, void *p_data, size_t *p_data_length)

Retrieve data associated with a provided uid.

Parameters

- **uid** (*psa_storage_uid_t*) – The uid value.
- **data_offset** (size_t) – The starting offset of the data requested.
- **data_size** (size_t) – The amount of data requested.
- **p_data** (void*) – On success, the buffer where the data will be placed.
- **p_data_length** (size_t*) – On success, will contain size of the data placed in p_data.

3.2.3.3.6.1 Description

Warning: Not supported

Retrieves up to data_size bytes of the data associated with uid, starting at data_offset bytes from the beginning of the data. Upon successful completion, the data will be placed in the p_data buffer, which must be at least data_size bytes in size. The length of the data returned will be in p_data_length. If data_size is 0, the contents of p_data_length will be set to zero.

3.2.3.3.6.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one of the provided arguments (e.g. p_data, p_data_length) is invalid, for example is NULL or references memory the caller cannot access. In addition, this can also happen if data_offset is larger than the size of the data associated with uid.
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided uid value was not found in the storage.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_GENERIC_ERROR:**
  The operation failed because of an unspecified internal failure.
- **PSA_ERROR_DATA_CORRUPT:**
  The operation failed because of an authentication failure when attempting to get the key.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The operation failed because the data associated with the uid failed authentication.

3.2.3.3.7 psa_ps_get_info

psa_status_t **psa_ps_get_info**(*psa_storage_uid_t* uid, struct *psa_storage_info_t* *p_info)

Retrieve the metadata about the provided uid.

Parameters

- **uid** (*psa_storage_uid_t*) – The identifier for the data.
- **p_info** (struct *psa_storage_info_t**) – A pointer to the psa_storage_info_t struct that will be populated with the metadata.

3.2.3.3.7.1 Description

Warning: Not supported

Retrieves the metadata stored for a given `uid` as a *struct psa_storage_info_t*.

3.2.3.3.7.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on).
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided `uid` value was not found in the storage.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_GENERIC_ERROR:**
  The operation failed because of an unspecified internal failure.
- **PSA_ERROR_DATA_CORRUPT:**
  The operation failed because of an authentication failure when attempting to get the key.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The operation failed because the data associated with the `uid` failed authentication.

3.2.3.3.8 psa_ps_remove

psa_status_t **psa_ps_remove**(*psa_storage_uid_t* uid)

Remove the provided uid and its associated data from the storage.

Parameters

- **uid** (*psa_storage_uid_t*) – The identifier for the data to be removed.

3.2.3.3.8.1 Description

Warning: Not supported

Removes previously stored data and any associated metadata, including rollback protection data.

3.2.3.3.8.2 Return

- **PSA_SUCCESS:**
  The operation completed successfully.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on).
- **PSA_ERROR_DOES_NOT_EXIST:**
  The operation failed because the provided `uid` value was not found in the storage.
- **PSA_ERROR_NOT_PERMITTED:**
  The operation failed because the provided `uid` value was created with PSA_STORAGE_FLAG_WRITE_ONCE.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_GENERIC_ERROR:**
  The operation failed because of an unspecified internal failure.

3.2.3.3.9 psa_ps_create

psa_status_t **psa_ps_create**(*psa_storage_uid_t* uid, size_t capacity, *psa_storage_create_flags_t* create_flags)

Create an asset based on parameters.

Parameters

- **uid** (*psa_storage_uid_t*) – A unique identifier for the asset.
- **capacity** (`size_t`) – The allocated capacity, in bytes, of the `uid`.
- **create_flags** (*psa_storage_create_flags_t*) – Flags indicating properties of the storage.

3.2.3.3.9.1 Description

Warning: Not supported

Reserves storage for the specified `uid`. Upon success, the capacity of the storage is `capacity`, and the size is 0. It is only necessary to call this function for assets that will be written with the *psa_ps_set_extended()* function. If only *psa_ps_set()* is needed, calls to this function are redundant.

This function cannot be used to replace an existing asset, and attempting to do so will return PSA_ERROR_ALREADY_EXISTS.

If the PSA_STORAGE_FLAG_WRITE_ONCE flag is passed, *psa_ps_create()* will return PSA_ERROR_NOT_SUPPORTED.

This function is supported only if the *psa_ps_get_support()* returns PSA_STORAGE_SUPPORT_SET_EXTENDED.

3.2.3.3.9.2 Return

- **PSA_SUCCESS:**
  The storage was successfully reserved.
- **PSA_ERROR_STORAGE_FAILURE:**
  The operation failed because the physical storage has failed (Fatal error).
- **PSA_ERROR_INSUFFICIENT_STORAGE:**
  `capacity` is bigger than the current available space.
- **PSA_ERROR_NOT_SUPPORTED:**
  The function is not implemented or one or more `create_flags` are not supported.
- **PSA_ERROR_INVALID_ARGUMENT:**
  `uid` was 0 or `create_flags` specified flags that are not defined in the API.
- **PSA_ERROR_GENERIC_ERROR:**
  The operation has failed due to an unspecified error.
- **PSA_ERROR_ALREADY_EXISTS:**
  Storage for the specified `uid` already exists.

3.2.3.3.10 psa_ps_set_extended

psa_status_t **psa_ps_set_extended**(*psa_storage_uid_t* uid, size_t data_offset, size_t data_length, const void *p_data)

Set partial data into an asset based on parameters

Parameters

- **uid** (*psa_storage_uid_t*) – The unique identifier for the asset.
- **data_offset** (`size_t`) – Offset within the asset to start the write.
- **data_length** (`size_t`) – The size in bytes of the data in `p_data` to write.
- **p_data** (`const void*`) – Pointer to a buffer which contains the data to write.

3.2.3.3.10.1 Description

Warning: Not supported

Sets partial data into an asset based on the given `uid`, `data_offset`, `data_length` and `p_data`.

Before calling this function, the storage must have been reserved with a call to *psa_ps_create()*. It can also be used to overwrite data in an asset that was created with a call to *psa_ps_set()*.

Calling this function with `data_length` = 0 is permitted. This makes no change to the stored data.

This function can overwrite existing data and/or extend it up to the capacity for the `uid` specified in psa_ps_create, but cannot create gaps. That is, it has preconditions:

- data_offset <= size
- data_offset + data_length <= capacity

and postconditions:

- size = max(size, data_offset + data_length)
- capacity unchanged.

This function is supported only if the *psa_ps_get_support()* returns PSA_STORAGE_SUPPORT_SET_EXTENDED.

3.2.3.3.10.2 Return

- **PSA_SUCCESS:**
  The asset exists, the input parameters are correct and the data is correctly written in the physical storage.
- **PSA_ERROR_STORAGE_FAILURE:**
  The data was not written correctly in the physical storage.
- **PSA_ERROR_INVALID_ARGUMENT:**
  The operation failed because one or more of the preconditions listed above regarding `data_offset`, size, or `data_length` was violated.
- **PSA_ERROR_DOES_NOT_EXIST:**
  The specified `uid` was not found.
- **PSA_ERROR_NOT_SUPPORTED:**
  The implementation of the API does not support this function.
- **PSA_ERROR_GENERIC_ERROR:**
  The operation failed due to an unspecified error.
- **PSA_ERROR_DATA_CORRUPT:**
  The operation failed because the existing data has been corrupted.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The operation failed because the existing data failed authentication (MAC check failed).
- **PSA_ERROR_NOT_PERMITTED:**
  The operation failed because it was attempted on an asset which was written with the flag PSA_STORAGE_FLAG_WRITE_ONCE.

3.2.3.3.11 psa_ps_get_support

uint32_t **psa_ps_get_support**(void)

Get implemented optional features

Parameters

- **void** – no arguments

3.2.3.3.11.1 Description

Warning: Not supported

Returns a bitmask with flags set for all of the optional features supported by the implementation.

Currently defined flags are limited to:

- PSA_STORAGE_SUPPORT_SET_EXTENDED

3.2.3.3.11.2 Return

uint32_t

3.2.4 Return codes

3.2.4.1 Introduction

This file defines the error codes returned by the PSA Cryptography API

3.2.4.2 Reference

Documentation:
PSA Cryptography API v1.1.0

Link:
https://developer.arm.com/documentation/ihi0086/b

3.2.4.3 typedef psa_status_t

type **psa_status_t**

Function return status.

3.2.4.3.1 Description

This is either PSA_SUCCESS, which is zero, indicating success; or a small negative value indicating that an error occurred. Errors are encoded as one of the PSA_ERROR_xxx values defined here.

3.2.4.3.2 Values

- **PSA_SUCCESS:**
  The action was completed successfully.
- **PSA_ERROR_ALREADY_EXISTS:**
  Asking for an item that already exists.

  It is recommended that implementations return this error code when attempting to write to a location where a key is already present.
- **PSA_ERROR_BAD_STATE:**
  The requested action cannot be performed in the current state.

  Multi-part operations return this error when one of the functions is called out of sequence. Refer to the function descriptions for permitted sequencing of functions.

  Implementations must not return this error code to indicate that a key identifier is invalid, but must return PSA_ERROR_INVALID_HANDLE instead.
- **PSA_ERROR_BUFFER_TOO_SMALL:**
  An output buffer is too small.

  Applications can call the PSA_xxx_SIZE macro listed in the function description to determine a sufficient buffer size.

  It is recommended that implementations only return this error code in cases when performing the operation with a larger output buffer would succeed. However, implementations can also return this error if a function has invalid or unsupported parameters in addition to an insufficient output buffer size.
- **PSA_ERROR_COMMUNICATION_FAILURE:**
  There was a communication failure inside the implementation.

  This can indicate a communication failure between the application and an external cryptoprocessor or between the cryptoprocessor and an external volatile or persistent memory. A communication failure can be transient or permanent depending on the cause.

  Warning:
  If a function returns this error, it is undetermined whether the requested action has completed. Returning PSA_SUCCESS is recommended on successful completion whenever possible, however functions can return PSA_ERROR_COMMUNICATION_FAILURE if the requested action was completed successfully in an external cryptoprocessor but there was a breakdown of communication before the cryptoprocessor could report the status to the application.
- **PSA_ERROR_CORRUPTION_DETECTED:**
  A tampering attempt was detected.

  If an application receives this error code, there is no guarantee that previously accessed or computed data was correct and remains confidential. In this situation, it is recommended that applications perform no further security functions and enter a safe failure state.

Implementations can return this error code if they detect an invalid state that cannot happen during normal operation and that indicates that the implementation's security guarantees no longer hold. Depending on the implementation architecture and on its security and safety goals, the implementation might forcibly terminate the application.

This error code is intended as a last resort when a security breach is detected and it is unsure whether the keystore data is still protected. Implementations must only return this error code to report an alarm from a tampering detector, to indicate that the confidentiality of stored data can no longer be guaranteed, or to indicate that the integrity of previously returned data is now considered compromised. Implementations must not use this error code to indicate a hardware failure that merely makes it impossible to perform the requested operation, instead use PSA_ERROR_COMMUNICATION_FAILURE, PSA_ERROR_STORAGE_FAILURE, PSA_ERROR_HARDWARE_FAILURE, PSA_ERROR_INSUFFICIENT_ENTROPY or other applicable error code.

This error indicates an attack against the application. Implementations must not return this error code as a consequence of the behavior of the application itself.

- **PSA_ERROR_DATA_CORRUPT:**
  Stored data has been corrupted.

  This error indicates that some persistent storage has suffered corruption. It does not indicate the following situations, which have specific error codes:

  - A corruption of volatile memory - use PSA_ERROR_CORRUPTION_DETECTED.
  - A communication error between the cryptoprocessor and its external storage - use PSA_ERROR_COMMUNICATION_FAILURE.
  - When the storage is in a valid state but is full - use PSA_ERROR_INSUFFICIENT_STORAGE.
  - When the storage fails for other reasons - use PSA_ERROR_STORAGE_FAILURE.
  - When the stored data is not valid - use PSA_ERROR_DATA_INVALID.

  Note that a storage corruption does not indicate that any data that was previously read is invalid. However this previously read data might no longer be readable from storage.

  When a storage failure occurs, it is no longer possible to ensure the global integrity of the keystore. Depending on the global integrity guarantees offered by the implementation, access to other data might fail even if the data is still readable but its integrity cannot be guaranteed.

  It is recommended to only use this error code to report when a storage component indicates that the stored data is corrupt, or fails an integrity check. For example, in situations that the PSA Storage API [PSA-ITS] reports PSA_ERROR_DATA_CORRUPT or PSA_ERROR_INVALID_SIGNATURE.

- **PSA_ERROR_DATA_INVALID:**
  Data read from storage is not valid for the implementation.

  This error indicates that some data read from storage does not have a valid format. It does not indicate the following situations, which have specific error codes:

  - When the storage or stored data is corrupted - use PSA_ERROR_DATA_CORRUPT.
  - When the storage fails for other reasons - use PSA_ERROR_STORAGE_FAILURE.
  - An invalid argument to the API - use PSA_ERROR_INVALID_ARGUMENT.

This error is typically a result of an integration failure, where the implementation reading the data is not compatible with the implementation that stored the data.

It is recommended to only use this error code to report when data that is successfully read from storage is invalid.

- **PSA_ERROR_DOES_NOT_EXIST:**
  Asking for an item that doesn't exist.

  Implementations must not return this error code to indicate that a key identifier is invalid, but must return PSA_ERROR_INVALID_HANDLE instead.

- **PSA_ERROR_GENERIC_ERROR:**
  An error occurred that does not correspond to any defined failure cause.

  Implementations can use this error code if none of the other standard error codes are applicable.

- **PSA_ERROR_HARDWARE_FAILURE:**
  A hardware failure was detected.

  A hardware failure can be transient or permanent depending on the cause.

- **PSA_ERROR_INSUFFICIENT_DATA:**
  Return this error when there's insufficient data when attempting to read from a resource.

- **PSA_ERROR_INSUFFICIENT_ENTROPY:**
  There is not enough entropy to generate random data needed for the requested action.

  This error indicates a failure of a hardware random generator. Application writers must note that this error can be returned not only by functions whose purpose is to generate random data, such as key, IV or nonce generation, but also by functions that execute an algorithm with a randomized result, as well as functions that use randomization of intermediate computations as a countermeasure to certain attacks.

  It is recommended that implementations do not return this error after `psa_crypto_init()` has succeeded. This can be achieved if the implementation generates sufficient entropy during initialization and subsequently a cryptographically secure pseudorandom generator (PRNG) is used. However, implementations might return this error at any time, for example, if a policy requires the PRNG to be reseeded during normal operation.

- **PSA_ERROR_INSUFFICIENT_MEMORY:**
  There is not enough runtime memory.

  If the action is carried out across multiple security realms, this error can refer to available memory in any of the security realms.

- **PSA_ERROR_INSUFFICIENT_STORAGE:**
  There is not enough persistent storage.

  Functions that modify the key storage return this error code if there is insufficient storage space on the host media. In addition, many functions that do not otherwise access storage might return this error code if the implementation requires a mandatory log entry for the requested action and the log storage space is full.

- **PSA_ERROR_INVALID_ARGUMENT:**
  The parameters passed to the function are invalid.

  Implementations can return this error any time a parameter or combination of parameters are recognized as invalid.

Implementations must not return this error code to indicate that a key identifier is invalid, but must return PSA_ERROR_INVALID_HANDLE instead.

- **PSA_ERROR_INVALID_HANDLE:**
  The key identifier is not valid.
- **PSA_ERROR_INVALID_PADDING:**
  The decrypted padding is incorrect.

  Warning:
  In some protocols, when decrypting data, it is essential that the behavior of the application does not depend on whether the padding is correct, down to precise timing. Protocols that use authenticated encryption are recommended for use by applications, rather than plain encryption. If the application must perform a decryption of unauthenticated data, the application writer must take care not to reveal whether the padding is invalid.

  Implementations must handle padding carefully, aiming to make it impossible for an external observer to distinguish between valid and invalid padding. In particular, it is recommended that the timing of a decryption operation does not depend on the validity of the padding.
- **PSA_ERROR_INVALID_SIGNATURE:**
  The signature, MAC or hash is incorrect.

  Verification functions return this error if the verification calculations completed successfully, and the value to be verified was determined to be incorrect.

  If the value to verify has an invalid size, implementations can return either PSA_ERROR_INVALID_ARGUMENT or PSA_ERROR_INVALID_SIGNATURE.
- **PSA_ERROR_NOT_PERMITTED:**
  The requested action is denied by a policy.

  It is recommended that implementations return this error code when the parameters are recognized as valid and supported, and a policy explicitly denies the requested operation.

  If a subset of the parameters of a function call identify a forbidden operation, and another subset of the parameters are not valid or not supported, it is unspecified whether the function returns PSA_ERROR_NOT_PERMITTED, PSA_ERROR_NOT_SUPPORTED or PSA_ERROR_INVALID_ARGUMENT.
- **PSA_ERROR_NOT_SUPPORTED:**
  The requested operation or a parameter is not supported by this implementation.

  It is recommended that implementations return this error code when an enumeration parameter such as a key type, algorithm, etc. is not recognized. If a combination of parameters is recognized and identified as not valid, return PSA_ERROR_INVALID_ARGUMENT instead.
- **PSA_ERROR_STORAGE_FAILURE:**
  There was a storage failure that might have led to data loss.

  This error indicates that some persistent storage could not be read or written by the implementation. It does not indicate the following situations, which have specific error codes:

  - A corruption of volatile memory - use PSA_ERROR_CORRUPTION_DETECTED.
  - A communication error between the cryptoprocessor and its external storage - use PSA_ERROR_COMMUNICATION_FAILURE.

- When the storage is in a valid state but is full - use PSA_ERROR_INSUFFICIENT_STORAGE.
- When the storage or stored data is corrupted - use PSA_ERROR_DATA_CORRUPT.
- When the stored data is not valid - use PSA_ERROR_DATA_INVALID.

A storage failure does not indicate that any data that was previously read is invalid. However this previously read data might no longer be readable from storage.

When a storage failure occurs, it is no longer possible to ensure the global integrity of the keystore. Depending on the global integrity guarantees offered by the implementation, access to other data might fail even if the data is still readable but its integrity cannot be guaranteed.

It is recommended to only use this error code to report a permanent storage corruption. However application writers must keep in mind that transient errors while reading the storage might be reported using this error code.

CHAPTER

FOUR

SUBSYSTEMS CAPABILITIES

4.1 ELE capabilities

4.1.1 Key manager

Table 4.1: ELE Key type

Key type	Key security size(s)
AES	128 / 192 / 256
ECDSA NIST	224 / 256 / 384 / 521
ECDSA BR1	224 / 256 / 384
HMAC	224 / 256 / 384 / 512

Operations supported:

- Generate
- Import (only EdgeLock 2GO object)
- Export (only public key in HEX or Base64 format)
- Delete
- Get key attributes
- Get key buffers' length
- Get key security size
- Get key type name
- Commit key storage

Key group: The SMW Library is managing the ELE key group automatically. The library is selecting a key group depending if a key is persistent/permanent or transient.

- Persistent/Permanent keys are in key groups from 0 to 49.
- Transient keys are in key groups from 50 to 99.

4.1.1.1 Key policy

When creating a new key, the key policy must be specified through the operation key attributes list. The key policy definition is defined with a **POLICY** TLV *Variable Length list*.

The following Table 4.2 lists all key usages applicable in ELE subsystem. A key policy defines one or more key usage.

Table 4.2: ELE Key usages

USAGE	Description
ENCRYPT	Permission to encrypt a message
DECRYPT	Permission to decrypt a message
SIGN_MESSAGE	Permission to sign a message
SIGN_HASH	Permission to sign a message hashed
VERIFY_MESSAGE	Permission to verify the signature of a message
VERIFY_HASH	Permission to verify the signature of message hashed
DERIVE	Permission to derive other keys from this key

The following Table 4.3 lists all permitted algorithms applicable in ELE subsystem. Only one permitted algorithm is allowed per key.

The key permitted algorithm definition:

- can be defines once with one of the key usages or repeated to each key usage.
- if more than one permitted algorithm is given in the key policy (one different per key usage or several per key usage), only the first algorithm is retained, others are ignored.

Table 4.3: ELE Key permitted algorithm

TLV Type			Comment
ALGO	**HASH**	**MIN_LENGTH LENGTH**	
HMAC	SHA256	From 8 to 32 bytes	If not specified length is 32 bytes
	SHA384	From 8 to 48 bytes	If not specified length is 48 bytes
ECB_NO_PADDING	N/A	N/A	
CBC_NO_PADDING	N/A	N/A	
CTR	N/A	N/A	
ALL_CIPHER	N/A	N/A	Support all ciphers including CMAC
CCM	N/A	N/A	
ALL_AEAD	N/A	N/A	Support all AEAD
RSA_PKCS1V15	N/A	N/A	Support all hash
	SHA1	N/A	
	SHA224	N/A	
	SHA256	N/A	
	SHA384	N/A	
	SHA512	N/A	
RSA_PSS	N/A	N/A	Support all hash
	SHA1	N/A	
	SHA224	N/A	
	SHA256	N/A	
	SHA384	N/A	
	SHA512	N/A	
ECDSA	SHA224	N/A	
	SHA256	N/A	
	SHA384	N/A	
	SHA512	N/A	
CMAC	N/A	N/A	

4.1.2 Hash

Table 4.4: ELE Hash

Hash Algorithm
SHA224
SHA256
SHA384
SHA512

4.1.3 Signature

Table 4.5: ELE Signature

Signature Type	Key type	Key security size(s)	Hash algorithm
ECDSA	ECDSA NIST	224 / 256 / 384 / 521	SHA224 SHA256 SHA384 SHA512 None (Message hashed)
	ECDSA BRAINPOOL R1	224 / 256 / 384 / 521	SHA224 SHA256 SHA384 None (Message hashed)

Operations supported:

- Sign
- Verify

4.1.3.1 Sign operation

The following key policies must defined:

- Usage:
  - SIGN_MESSAGE to sign a message to be hashed
  - SIGN_HASH to sign a message already hashed
- Algorithm:
  - for an ECDSA Signature, ECDSA with any hash or a hash already as listed in Table 4.5

4.1.3.2 Verify operation

The following key policies must defined if a key identifier is used:

- Usage:
  - VERIFY_MESSAGE to verify the signature of a message to be hashed
  - VERIFY_HASH to verify the signature of a message already hashed
- Algorithm:
- ECDSA with any hash or a hash already as listed in Table 4.5

4.1.4 Random

Length: 1 to UINT32_MAX

4.1.5 MAC

Table 4.6: ELE MAC

Key type	Key security size(s)	Algorithm
AES	128 / 192 / 256	CMAC CMAC_TRUNCATED
HMAC	224 / 256 / 384 / 512	HMAC HMAC_TRUNCATED

The MAC size can be truncated if the key permitted algorithm limits the MAC output length.

Operations supported:

- Compute MAC
- Verify MAC

4.1.5.1 Compute MAC operation

MAC generation operation can compute either a full MAC length or a truncated MAC length. The operation algorithm and key permitted algorithm allows to select the MAC length to be generated.

Table 4.7: ELE MAC - Compute

MAC Length	Algorithm	Hash	Key policy
Full MAC	CMAC	N/A	Usage: SIGN_MESSAGE Algorithm: CMAC
	HMAC	SHA256 SHA384	Usage: SIGN_MESSAGE Algorithm: HMAC with HASH=[256/384]
Truncated MAC Minimum length	CMAC_TRUNCATED	N/A	Usage: SIGN_MESSAGE Algorithm: CMAC with MIN_LENGTH=[min]
	HMAC_TRUNCATED	SHA256 SHA384	Usage: SIGN_MESSAGE Algorithm: HMAC with HASH=[256/384] and MIN_LENGTH=[min]
Truncated MAC Fix length	CMAC_TRUNCATED	N/A	Usage: SIGN_MESSAGE Algorithm: CMAC with LENGTH=[length]
	HMAC_TRUNCATED	SHA256 SHA384	Usage: SIGN_MESSAGE Algorithm: HMAC with HASH=[256/384] and LENGTH=[min]

4.1.5.2 Verify MAC operation

MAC verification operation can verify either a full MAC length or a truncated MAC length. The operation algorithm and key permitted algorithm allows to select the MAC length to be generated.

Table 4.8: ELE MAC - Verify

MAC Length	Algorithm	Hash	Key policy
Full MAC	CMAC	N/A	Usage: VERIFY_MESSAGE Algorithm: CMAC
	HMAC	SHA256 SHA384	Usage: VERIFY_MESSAGE Algorithm: HMAC with HASH=[256/384]
Truncated MAC Minimum length	CMAC_TRUNCATED	N/A	Usage: VERIFY_MESSAGE Algorithm: CMAC with MIN_LENGTH=[min]
	HMAC_TRUNCATED	SHA256 SHA384	Usage: VERIFY_MESSAGE Algorithm: HMAC with HASH=[256/384] and MIN_LENGTH=[min]
Truncated MAC Fix length	CMAC_TRUNCATED	N/A	Usage: VERIFY_MESSAGE Algorithm: CMAC with LENGTH=[length]
	HMAC_TRUNCATED	SHA256 SHA384	Usage: VERIFY_MESSAGE Algorithm: HMAC with HASH=[256/384] and LENGTH=[min]

4.1.6 Cipher

Table 4.9: ELE Cipher

Key type	Mode
AES	CBC ECB CTR

One-shot operations supported:

- Encrypt

- Decrypt

4.1.6.1 Encrypt operation

The following key policies must defined:

- Usage: ENCRYPT
- Algorithm:
  - CBC_NO_PADDING
  - ECB_NO_PADDING
  - CTR
  - ALL_CIPHER (any cipher mode)

4.1.6.2 Decrypt operation

The following key policies must defined if a key identifier is used:

- Usage: DECRYPT
- Algorithm:
  - CBC_NO_PADDING
  - ECB_NO_PADDING
  - CTR
  - ALL_CIPHER (any cipher mode)

4.1.7 AEAD

Table 4.10: ELE AEAD

Key type	Mode
AES	CCM

One-shot operations supported:

- AEAD Encryption
- AEAD Decryption

4.1.8 Device management

The following operations are available:

- Device Attestation
- Device UUID (in big endian format)
- Device lifecycle

4.1.8.1 Device Attestation

The device attestation requires a challenge value to guaranty the certificate request. The challenge value maximum length depends of the device as listed in the following table.

Table 4.11: ELE Attestation Challenge

Device	Challenge Length in bytes
i.MX8ULP	4
i.MX93	16

4.1.8.2 Device lifecycle

The device lifecycle operations supported are get and set device lifecycle. The following table lists the device lifecycle supported when executing a get or set device lifecycle.

> **Warning:** Changing the device lifecycle (set operation) is not revertable. Refer to the device documentation to get more details about the lifecycle.

Table 4.12: ELE Device lifecycle

Lifecycle	Get	Set	Comment
OPEN	Yes	Yes	
CLOSED	Yes	Yes	A signed image is required to boot
CLOSED_LOCKED	Yes	Yes	A signed image is required to boot
OEM_RETURN	No	Yes	Device is no more OEM usable and must be returned to NXP
NXP_RETURN	No	Yes	Device is no more usable and must be returned to NXP

4.1.9 Data Storage manager

Data Storage manager allows to store and retreive data. The data ID is a 32-bits value with the exception of the 0xF00000E0 reserved for EdgeLock 2GO claimcode.

The subsystem allows to:

- store and retreive user data.
- encrypt and sign data (Table 4.13) before storing it and retreive a TLV blob (Table 4.14).
- set encypted and signed data as READ_ONCE, meaning that when data is retreived the subsystem deletes the data.

The subsystem doesn't allow to:

- delete a data.

Notes:

- Data size is limited to 2048 bytes.
- Data size must be aligned on a cipher block in case of data encryption. in other word, user must pad to the data.
- Data lifecycle can be defined only when storing encrypted/signed data.

Table 4.13: ELE Data Encrypt/Sign

Encryption	IV	Signature
ECB_NO_PADDING	N/A	CMAC
CBC_NO_PADDING	Yes	
CTR	Yes	
CFB	Yes	

Table 4.14: ELE Data blob (encrypted and signed)

Tag	Length (bytes)	Value/Description
0x41	16	Device UUID in big endian format.
0x45	16	Value of the IV used to encrypt data in case encryption algorithm use an IV. The IV can be either given as input by the user or randomly generated by the subsystem (user must the IV buffer and its length to 0).
0x46	Variable	Encrypted data. Maximum length is 2048 bytes.

4.2 HSM capabilities

4.2.1 Key manager

Table 4.15: HSM Key type

Key type	Key security size(s)
AES	128 / 192 / 256
ECDSA BR1	256 / 384
ECDSA NIST	256 / 384
HMAC_SHA224	224
HMAC_SHA256	256
HMAC_SHA384	384
HMAC_SHA512	512

Operations supported:

- Generate
- Export (only public key in HEX or Base64 format)
- Delete
- Derive[1]
- Get key attributes
- Get key buffers' length
- Get key security size
- Get key type name
- Commit key storage (do nothing)

Key group: The SMW Library is managing the HSM key group automatically. The library is selecting a key group depending if a key is persistent/permanent or transient.

- Persistent/Permanent keys are in key groups from 0 to 511.
- Transient keys are in key groups from 512 to 1023.

Persistent key: To flush persistent key, "FLUSH_KEY" attribute must be set. When set, HSM executes a strict operation and all keys defined as persistent are flushed. Note that HSM uses a strict operation counter which is a replay attack counter, then the number of strict operation is limited. So when possible it's better to perform multiple persistent key operations (generate, import, delete) before setting the "FLUSH_KEY" attribute.

[1] Only TLS12_KEY_EXCHANGE when hardware supports it

4.2.1.1 Key policy

The HSM subsystem doesn't support key policy attribute. Defining the key attribute **POLICY** will be ignored and if attribute is defined the API returns the warning *SMW_STATUS_KEY_POLICY_WARNING_IGNORED*.

4.2.2 Hash

Table 4.16: HSM Hash

Hash Algorithm
SHA224
SHA256
SHA384
SHA512

4.2.3 Signature

Table 4.17: HSM Signature

Key type	Key security size(s)	Hash algorithm
ECDSA BR1	256	SHA256
	384	SHA384
ECDSA NIST	256	SHA256
	384	SHA384

Operations supported:

- Sign[2]
- Verify

4.2.4 Random

Length: 1 to UINT32_MAX

[2] Attribute TLS_MAC_FINISH available only when hardware supports it

4.2.5 MAC

Table 4.18: HSM MAC

Key type	Key security size(s)	Algorithm	Hash
AES	128 / 192 / 256	CMAC	N/A
HMAC_SHA224	224	HMAC	SHA224
HMAC_SHA256	256	HMAC	SHA256
HMAC_SHA384	384	HMAC	SHA384
HMAC_SHA512	512	HMAC	SHA512

HMAC Key generation and HMAC generation is not working on all HSM Firmware and may return `SMW_STATUS_SUBSYSTEM_FAILURE`.

Operations supported:

- Compute MAC
- Verify MAC

4.2.6 Cipher

Table 4.19: HSM Cipher

Key type	Mode
AES	CBC ECB

One-shot operations supported:

- Encrypt
- Decrypt

4.2.7 AEAD

Table 4.20: HSM AEAD

Key type	Mode
AES	CCM GCM

One-shot operations supported:

- AEAD Encryption

- AEAD Decryption

4.3 TEE capabilities

4.3.1 Key manager

Table 4.21: TEE Key type

Key type	Key security size(s)
AES	128 / 192 / 256
DES	56
DES3	112 / 168
ECDSA NIST	192 / 224 / 256 / 384 / 521
RSA	256 to 4096[1]
HMAC_MD5	64 to 512 bits[2]
HMAC_SHA1	80 to 512 bits[2]
HMAC_SHA224	112 to 512 bits[2]
HMAC_SHA256	192 to 1024 bits[2]
HMAC_SHA384	256 to 1024 bits[2]
HMAC_SHA512	256 to 1024 bits[2]
HMAC_SM3	80 to 1024 bits[2]

Operations supported:

- Generate
- Import
- Export (only public key in HEX or Base64 format)
- Delete
- Get key attributes
- Get key buffers' length
- Get key security size
- Get key type name
- Commit key storage (do nothing)

[1] multiple of 2 bits
[2] multiple of 8 bits

4.3.1.1 Key policy

When creating a new key, the key policy must be specified through the operation key attributes list. The key policy definition is defined with a **POLICY** TLV *Variable Length list*.

The following Table 4.22 lists all key usages applicable in TEE subsystem. A key policy defines one or more key usage.

Table 4.22: TEE Key usages

USAGE	Description
ENCRYPT	Permission to encrypt a message
DECRYPT	Permission to decrypt a message
SIGN_MESSAGE	Permission to sign a message
SIGN_HASH	Permission to sign a message hashed
VERIFY_MESSAGE	Permission to verify the signature of a message
VERIFY_HASH	Permission to verify the signature of message hashed
DERIVE	Permission to derive other keys from this key
EXPORT	Permission to export the public key only

The TEE subsystem doesn't define algorithm restriction per key usage. Defining permitted algorithm(s) will not be taken into account and operation will return the warning status *SMW_STATUS_KEY_POLICY_WARNING_IGNORED*.

Caution: If key attribute **POLICY** is not specified, all key usages listed in the Table 4.22 are attributed to the created key.

4.3.2 Hash

Table 4.23: TEE Hash

Hash Algorithm
MD5
SHA1
SHA224
SHA256
SHA384
SHA512
SM3

4.3.3 Signature

Table 4.24: TEE Signature

Key type	Key security size(s)	Hash	Signature type
ECDSA NIST	192 / 224 / 256 / 384 / 521	SHA224 SHA256 SHA384 SHA512	N/A
RSA	256 to 4096[3]	MD5 SHA1 SHA224 SHA256 SHA384	RSASSA-PKCS1-V1_5 RSASSA-PSS

Operations supported:

- Sign
- Verify

4.3.4 MAC

Table 4.25: TEE MAC

Key type	Key security size(s)	Algorithm	Hash
AES	128 / 192 / 256	CMAC	N/A
HMAC_MD5	64 to 512 bits[4]	HMAC	MD5
HMAC_SHA1	80 to 512 bits[4]	HMAC	SHA1
HMAC_SHA224	112 to 512 bits[4]	HMAC	SHA224
HMAC_SHA256	192 to 1024 bits[4]	HMAC	SHA256
HMAC_SHA384	256 to 1024 bits[4]	HMAC	SHA384
HMAC_SHA512	256 to 1024 bits[4]	HMAC	SHA512
HMAC_SM3	80 to 1024 bits[4]	HMAC	SM3

Operations supported:

- Compute MAC

[3] multiple of 2 bits
[4] multiple of 8 bits

- Verify MAC

4.3.5 Random

Length: 1 to SIZE_MAX

4.3.6 Cipher

Table 4.26: TEE Cipher

Key type	Mode
AES	CBC CTR CTS ECB XTS
DES	CBC ECB
DES3	CBC ECB

Operations supported:

- Encrypt[5]
- Decrypt[Page 317, 5]

4.3.7 Operation context

Operations supported:

- Cancel
- Copy

[5] one shot and multi-part

4.3.8 AEAD

Table 4.27: TEE AEAD

Key type	Mode
AES	CCM GCM

Operations supported:

- Encrypt[6]
- Decrypt[Page 318, 6]

[6] one shot and multi-part

CHAPTER

FIVE

TLV CODING

The TLV is a Type-Length-Value coding scheme. Encoding data with this format allows to define optional information with variable length in a non-sorted list of data.

In the context of the Security Middleware library, the TLV data format is used to pass optional operation parameter(s) without dedicated operation argument structure field(s).

The **Type** is encoded as an ASCII string terminated with the null character. The possible values are specific to each operation.

The **Length** field is length in bytes of the **Value** field encoded with two bytes (MSB first). A length of 0 implies that **Value** field is not present.

The **Value** field is a byte stream that contains the data. Different type of data are supported: null terminated string, numeral, variable length list (TLV encoded list).

The Fig. 5.1 shows the binary translation of any type of TLVs.

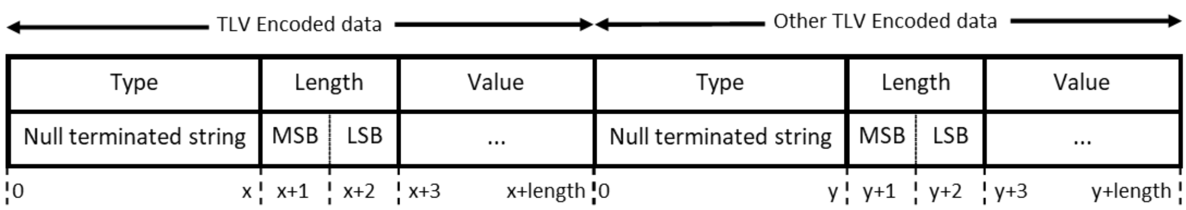

Fig. 5.1: Binary view of TLV encoded data

The TLV types are described in the following sections.

5.1 Boolean

5.1.1 Definition

As shown in the figure Fig. 5.2 below, a **boolean** is encoded with:

- *Type* is the name of the boolean to enable.
- *Length* always equals 0.
- *Value* is not present.

Defining a TLV boolean corresponds to set the boolean named by the type to *True*. A boolean can't be set to *False* explicitly in TLV. In other words, to set a boolean to *False*, it must not be defined in the TLV.

If the length is not 0, an error is returned.

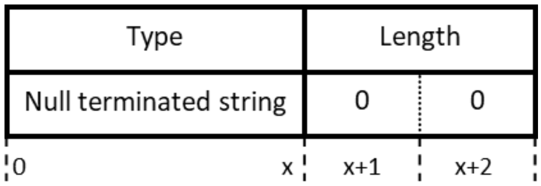

Fig. 5.2: TLV boolean data

5.1.2 Example

The Table 5.1 is the coding of the boolean attribute named *PERSISTENT*. When present in the operation attribute lists, the key persistency is enabled.

Table 5.1: Example of TLV boolean

	Type	Length
Data	PERSISTENT	0
Hex	0x50 0x45 0x52 0x53 0x49 0x53 0x54 0x45 0x4E 0x54 0x00	0x00 0x00

5.2 Numeral

5.2.1 Definition

As shown in the figure Fig. 5.3 below, a **numeral** is encoded with:

- *Type* is the name of the numeral to set.
- *Length* is the number of bytes of *Value*.
- *Value* is the numeric value in big-endian format.

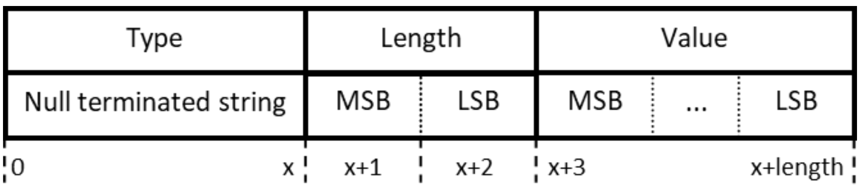

Fig. 5.3: TLV numeral data

Two categories of numeral are defined (see Table 5.2); the C standard type and the large numeral that is a hexdecimal buffer.

Table 5.2: TLV Numeral type

Numeral	Length
byte	1
short	2
integer	4
long long	8
large numeral	> 0

5.2.2 Examples

The Table 5.3 is the coding of a short integer attribute named *COUNTER* set to 500.

Table 5.3: Example of TLV short value

	Type	Length	Value
Data	COUNTER	2	500
Hex	0x43 0x4F 0x55 0x4E 0x54 0x45 0x52 0x00	0x00 0x02	0x01 0xF4

The Table 5.4 is the coding of a large integer attribute named *RSA_PUB_EXP* set to 1,180,591,621,000,000,000,001.

Table 5.4: Example of TLV large value

	Type	Length	Value
Data	RSA_PUB_EXP	9	1,180,591,621,000,000,000,001
Hex	0x52 0x53 0x41 0x5F 0x50 0x55 0x42 0x5F 0x45 0x58 0x50 0x00	0x00 0x09	0x40 0x00 0x00 0x00 0x41 0xCB 0x99 0x50 0x01

5.3 String

5.3.1 Definition

As shown in the figure Fig. 5.4 below, a **string** is encoded with:

- *Type* is the name of the string to set.
- *Length* is the number of bytes of *Value*.
- *Value* is the null terminated string value.

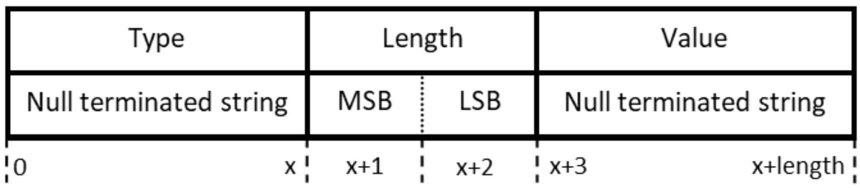

Fig. 5.4: TLV string data

5.3.2 Example

The Table 5.5 is the coding of a string attribute named *USER_NAME* set to “John Doe”.

Table 5.5: Example of TLV string value

	Type	Length	Value
Data	USER_NAME	9	John Doe
Hex	0x55 0x53 0x45 0x52 0x5F 0x4E 0x41 0x4D 0x45 0x00	0x00 0x09	0x4A 0x6F 0x68 0x6E 0x20 0x44 0x6F 0x65 0x00

5.4 Variable Length list

5.4.1 Definition

As shown in the figure Fig. 5.5 below, a **variable-length** is encoded with:

- *Type* is the name of the variable to set.
- *Length* is the number of bytes of *Value*.
- *Value* is a concatenation of one null terminated string and byte streams. The byte streams is a suite of TLV encoded data.

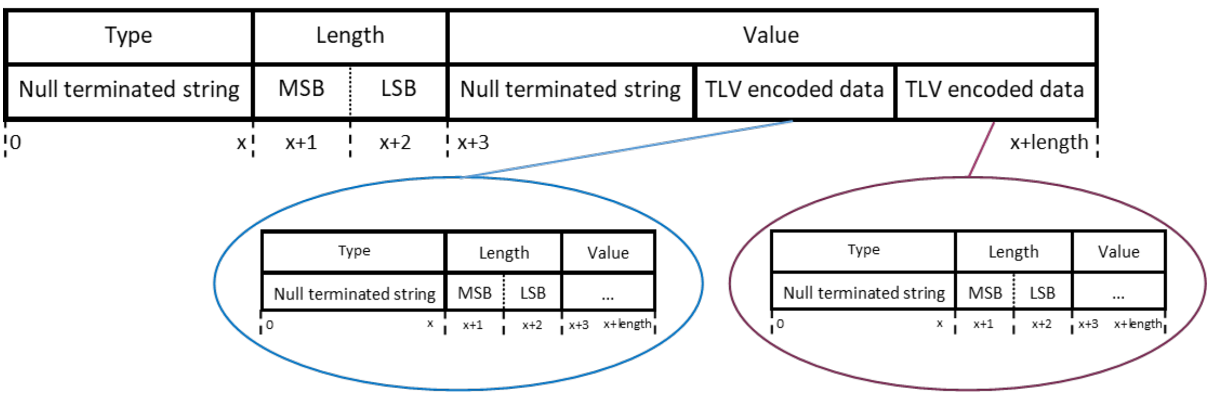

Fig. 5.5: TLV variable-length data

5.4.2 Example

The Table 5.6 is the coding of key policies attribute using a TLV variable-length specific coding. The key policies attribute tag type is *POLICY*, the length is the number of bytes of the *Value* field that is a variable-length list.

The example is encoding the key policies with usages:

a) Copiable (USAGE_COPY)

b) Encryption with restricted algorithms (USAGE_ENCRYPTION):

- Cipher with CBC mode with minimum tag length equal to 32 bits.
- Cipher Authenticated encryption with CCM mode without Tag length restriction (all tag lengths supported).

c) Signature generation with restricted algorithm (USAGE_SIGN):

- HMAC 256 bits.

Table 5.6: Example of TLV variable-length value

Type = POLICY	Length	Value
0x50 0x4F 0x4C 0x49 0x43 0x59 0x00	0x00 0x70	0x55 0x53 0x41 0x47 0x45 0x00 0x00 0x05 0x43 0x4F 0x50 0x59 0x00 0x55 0x53 0x41 0x47 0x45 0x00 0x00 0x37 0x45 0x4E 0x43 0x52 0x59 0x20 0x54 0x00 0x41 0x4C 0x47 0x4F 0x00 0x00 0x1D 0x43 0x42 0x43 0x5F 0x4E 0x4F 0x5F 0x50 0x41 0x44 0x44 0x49 0x4E 0x47 0x00 0x4D 0x49 0x4E 0x5F 0x4C 0x45 0x4E 0x47 0x54 0x48 0x00 0x00 0x01 0x20 0x41 0x4C 0x47 0x4F 0x00 0x00 0x04 0x43 0x43 0x4D 0x00 0x55 0x53 0x41 0x47 0x45 0x00 0x00 0x1F 0x53 0x49 0x47 0x4E 0x00 0x41 0x4C 0x47 0x4F 0x00 0x00 0x13 0x48 0x4D 0x41 0x43 0x00 0x48 0x41 0x53 0x48 0x00 0x00 0x07 0x53 0x48 0x41 0x32 0x35 0x36 0x00

Table 5.7: Details of key policies example Table 5.6

	Usage			Algo and parameter(s)		
	Type	Length	Value	Type	Length	Value
Data	USAGE	5	COPY			
Hex	0x55 0x53 0x41 0x47 0x45 0x00	0x00 0x05	0x43 0x4F 0x50 0x59 0x00			
Data	USAGE	55	ENCRYPT	ALGO	29	CBC_NO_PADDING
Hex	0x55 0x53 0x41 0x47 0x45 0x00	0x00 0x37	0x45 0x4E 0x43 0x52 0x59 0x50 0x54 0x00	0x41 0x4C 0x47 0x4F 0x00	0x00 0x1D	0x43 0x42 0x43 0x5F 0x4E 0x4F 0x5F 0x50 0x41 0x44 0x44 0x49 0x4E 0x47 0x00
Data				MIN_LENGTH	1	32
Hex				0x4D 0x49 0x4E 0x5F 0x4C 0x45 0x4E 0x47 0x54 0x48 0x00	0x00 0x01	0x20
Data				ALGO	4	CCM
Hex				0x41 0x4C 0x47 0x4F 0x00	0x00 0x04	0x43 0x43 0x4D 0x00
Data	USAGE	31	SIGN	ALGO	19	HMAC
*Hex**	0x55 0x53 0x41 0x47 0x45 0x00	0x00 0x1F	0x53 0x49 0x47 0x4E 0x00	0x41 0x4C 0x47 0x4F 0x00	0x00 0x13	0x48 0x4D 0x41 0x43 0x00
Data				HASH	7	SHA256
Hex				0x48 0x41 0x53 0x48 0x00	0x00 0x07	0x53 0x48 0x41 0x32 0x35 0x36 0x00

CHAPTER SIX

OSAL

In order to work on any Operating System (OS), the SMW library requires a module called OSAL. This OSAL must be implemented by the SMW library integrator to work on a specific OS.

The OSAL is the entry point of the SMW library. It's in charge of the library initialization, load, unload. In addition, the OSAL must implement a key database manager to convert a subsystem key identifier to a library identifier that might be PSA compatible or not.

The SMW source package contains an example of OSAL in the folder *osal/Linux* running on Linux and used to validate the library. This code example can be modified by the library integrator.

6.1 SMW interface

6.1.1 Introduction

The OSAL module must refer to the following structures and functions to be linked with the SMW core library. The content and prototype of the structures and functions can't be changed otherwise the core library may not build or work correctly.

6.1.2 struct osal_obj

struct **osal_obj**

OSAL object database operation parameters

6.1.2.1 Definition

```
struct osal_obj {
    unsigned int id;
    struct {
        unsigned int min;
        unsigned int max;
    } range;
    int persistence;
    void *info;
    size_t info_size;
}
```

6.1.2.2 Members

id
Object id output when object added, else input

range
Object id range to generate (information set by SMW at object creation)

range.min
Minimum value

range.max
Maximum value

persistence
Object persistence (information set by SMW at object creation)

info
Object information to store or restore

info_size
Size of the object information

6.1.2.3 Description

This structure defines the object information to be handled by the OSAL object database if needed.

6.1.2.4 Note

if object range min and max are equal, the object id is not generated by the object database manager.

6.1.3 struct smw_ops

struct **smw_ops**
SMW OSAL operations

6.1.3.1 Definition

```
struct smw_ops {
    void (*critical_section_start)(void);
    void (*critical_section_stop)(void);
    int (*mutex_init)(void **mutex);
    int (*mutex_destroy)(void **mutex);
    int (*mutex_lock)(void *mutex);
    int (*mutex_unlock)(void *mutex);
    int (*thread_create)(unsigned long *thread, void *(*start_routine)(void␣
↪*), void *arg);
    int (*thread_cancel)(unsigned long thread);
    void (*vprint)(const char *format, va_list arg);
    void (*hex_dump)(const unsigned char *addr, unsigned int size, unsigned␣
```

(continues on next page)

(continued from previous page)

```
↪int align);
    void (*register_active_subsystem)(const char *subsystem_name);
    int (*get_subsystem_info)(const char *subsystem_name, void *info);
    bool (*is_lib_initialized)(void);
    int (*get_obj_info)(struct osal_obj *obj);
    int (*add_obj_info)(struct osal_obj *obj);
    int (*update_obj_info)(struct osal_obj *obj);
    int (*delete_obj_info)(struct osal_obj *obj);
}
```

6.1.3.2 Members

critical_section_start
[optional] Start critical section

critical_section_stop
[optional] Stop critical section

mutex_init
[mandatory] Initialize a mutex

mutex_destroy
[mandatory] Destroy a mutex

mutex_lock
[mandatory] Lock a mutex

mutex_unlock
[mandatory] Unlock a mutex

thread_create
[mandatory] Create a thread

thread_cancel
[mandatory] Cancel a thread

vprint
[optional] Print debug trace

hex_dump
[optional] Print buffer content

register_active_subsystem
[optional] Register the active Secure Subsystem

get_subsystem_info
[mandatory] Get Subsystem configuration info

is_lib_initialized
[mandatory] Check if the library was successfully initialized by OSAL

get_obj_info
[mandatory] Get an object information from database

add_obj_info
[mandatory] Add an object information into database

update_obj_info
[mandatory] Update an object information into database

delete_obj_info
[mandatory] Delete an object information from database

6.1.3.3 Description

This structure defines the SMW OSAL. Functions pointers marked as [mandatory] must be assigned. Functions pointers marked as [optional] may not be assigned. mutex_* functions pointers are optional together. critical_* functions pointers are optional together.

6.1.4 smw_init

enum *smw_status_code* **smw_init**(const struct *smw_ops* *ops)

Initialize the SMW library.

Parameters

- **ops** (const struct *smw_ops**) – pointer to the structure describing the OSAL.

6.1.4.1 Description

This function initializes the Security Middleware. It verifies that ops is valid and then initializes SMW modules.

6.1.4.2 Return

See *enum smw_status_code*

- SMW_STATUS_OK - Initialization is successful
- SMW_STATUS_OPS_INVALID - ops is invalid
- SMW_STATUS_MUTEX_INIT_FAILURE - Mutex initialization has failed

6.1.5 smw_deinit

enum *smw_status_code* **smw_deinit**(void)

Deinitialize the SMW library.

Parameters

- **void** – no arguments

6.1.5.1 Description

This function deinitializes the Security Middleware. It frees all memory dynamically allocated by SMW.

6.1.5.2 Return

See *enum smw_status_code*

- SMW_STATUS_OK - Deinitialization is successful
- SMW_STATUS_INVALID_LIBRARY_CONTEXT - Library context is not valid
- SMW_STATUS_MUTEX_DESTROY_FAILURE - Mutex destruction has failed

6.2 Linux example

6.2.1 Introduction

The OSAL interface is the library API specific to the Operating System. It's under the charge of the library integrator to adapt the OSAL library part to the OS targeted.

Below is a C code example configuring and loading the SMW library with the given OSAL example.

```c
#define DEFAULT_OBJ_DB "/var/tmp/obj_db_smw_test.dat"

static const struct tee_info tee_default_info = {
    { "11b5c4aa-6d20-11ea-bc55-0242ac130003" }
};

static const struct se_info se_default_info = { 0x534d5754, 0x444546,
                                                1000 }; // SMWT, DEF

int main(int argc, char *argv[])
{
    int res = ERR_CODE(FAILED);

    // Configure the TEE Subsystem: TA UUID (and so key storage)
    res = smw_osal_set_subsystem_info("TEE", &tee_default_info,
                                      sizeof(tee_default_info));
    if (res != SMW_STATUS_OK)
        goto exit;

    // Configure the HSM Subsystem: Key storage identifier and replay
    res = smw_osal_set_subsystem_info("HSM", &se_default_info,
                                       sizeof(se_default_info));
    if (res != SMW_STATUS_OK)
        goto exit;

    // Open/Create the application object database
    res = smw_osal_open_obj_db(DEFAULT_OBJ_DB, strlen(DEFAULT_OBJ_DB) + 1);
    if (res != SMW_STATUS_OK)
```

(continues on next page)

(continued from previous page)

```
        goto exit;

    // Load and initialize the library. OSAL is loading the appplication
    // SMW configuration file defined by the system environment variable
    // 'SMW_CONFIG_FILE'
    res = smw_osal_lib_init();
    if (res != SMW_STATUS_OK)
        goto exit;

    // Execute the application
    ...

    exit:

    return res;
}
```

6.2.2 struct tee_info

struct **tee_info**

TEE Subsystem information

6.2.2.1 Definition

```
struct tee_info {
    char ta_uuid[TEE_TA_UUID_SIZE_MAX];
}
```

6.2.2.2 Members

ta_uuid
TA UUID

6.2.3 struct se_info

struct **se_info**

Secure Enclave information

6.2.3.1 Definition

```
struct se_info {
    unsigned int storage_id;
    unsigned int storage_nonce;
    unsigned short storage_replay;
}
```

6.2.3.2 Members

storage_id
Key storage identifier

storage_nonce
Key storage nonce

storage_replay
Replay attack counter (Not used on ELE)

6.2.4 smw_osal_latest_subsystem_name

const char ***smw_osal_latest_subsystem_name**(void)

Return the latest Secure Subsystem name

Parameters

- **void** – no arguments

6.2.4.1 Description

In DEBUG mode only, function returns the name of the latest Secure Subsystem invoked by SMW. This Secure Subsystem have been either explicitly requested by the caller or selected by SMW given the operation arguments and the configuration file. In other modes, function always returns NULL.

6.2.4.2 Return

In DEBUG mode only, the pointer to the static buffer containing the null-terminated string name of the Secure Subsystem. In other modes, NULL

6.2.5 smw_osal_lib_init

enum *smw_status_code* **smw_osal_lib_init**(void)

Initialize the SMW library

Parameters

- **void** – no arguments

6.2.5.1 Description

This function must be the first function called by the application opening a library instance. It loads the subsystem configuration set in the linux environment variable SMW_CONFIG_FILE.

6.2.5.2 Return

SMW_STATUS_OK - Library initialization success SMW_STATUS_LIBRARY_ALREADY_INIT - Library already initialized otherwise any of the smw status

6.2.6 smw_osal_set_subsystem_info

enum *smw_status_code* **smw_osal_set_subsystem_info**(*smw_subsystem_t* subsystem, void *info, size_t info_size)

Set the Subsystem configuration information

Parameters

- **subsystem** (*smw_subsystem_t*) – Subsystem name
- **info** (void*) – Subsystem information
- **info_size** (size_t) – Size in bytes of `info` parameter

6.2.6.1 Description

This function must be called before a subsystem is loaded.

6.2.6.2 Return

See *enum smw_status_code*

- SMW_STATUS_OK - Success
- SMW_STATUS_SUBSYSTEM_LOADED - Subsystem is already loaded
- SMW_STATUS_INVALID_PARAM - Function parameter error
- SMW_STATUS_ALLOC_FAILURE - Allocation failure
- SMW_STATUS_UNKNOWN_NAME - Subsystem unknown

6.2.7 smw_osal_open_key_db

enum *smw_status_code* **smw_osal_open_key_db**(const char *file, size_t len)

Open a key database file

Parameters

- **file** (const char*) – Fullname of the key database
- **len** (size_t) – Length of the `file` string

6.2.7.1 Description

Deprecated. Will be removed in library version 3.x. Use *smw_osal_open_obj_db()*.

6.2.7.2 Return

See ***enum smw_status_code***

- SMW_STATUS_OK - Success
- SMW_STATUS_KEY_DB_INIT - Initialization error of the database

6.2.8 smw_osal_open_obj_db

enum *smw_status_code* **smw_osal_open_obj_db**(const char *file, size_t len)

Open a object database file

Parameters

- **file** (const char*) – Fullname of the object database
- **len** (size_t) – Length of the file string

6.2.8.1 Return

See ***enum smw_status_code***

- SMW_STATUS_OK - Success
- SMW_STATUS_OBJ_DB_INIT - Initialization error of the database

INDEX

O

P

S

T