PSA Certified Secure Storage API 1.0

Document number:	IHI 0087
Release Quality:	Final
Issue Number:	2
Confidentiality:	Non-confidential
Date of Issue:	23/03/2023

Contents

About this document

Release information

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

Table 1 Document revision history

Date	Version	Confidentiality	Change
Feb 2019	1.0 beta 2	Non-confidential	Initial publication.
June 2019	1.0.0	Non-confidential	First stable release with 1.0 API finalized. Uses the common PSA Certified Status codes. Modified the API parameters to align with other PSA Certified APIs. Added storage flags to specify protection requirement.
October 2022	1.0.1	Non-confidential	Relicensed as open source under CC BY-SA 4.0. Documentation clarifications.
March 2023	1.0.2	Non-confidential	Documentation clarifications.

The detailed changes in each release are described in *Document history* on page 33.

PSA Certified Secure Storage API

License

Text and illustrations

About the license

The language in the additional patent license is largely identical to that in section 3 of the Apache License, Version 2.0 (Apache 2.0), with two exceptions:

1. Changes are made related to the defined terms, to align those defined terms with the terminology in CC BY-SA 4.0 rather than Apache 2.0 (for example, changing "Work" to "Licensed Material").
2. The scope of the defensive termination clause is changed from "any patent licenses granted to You" to "any licenses granted to You". This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

To view the full text of the Apache 2.0 license, visit apache.org/licenses/LICENSE-2.0.

Source code

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[PSM]	ARM DEN 0128	*Platform Security Model*. developer.arm.com/documentation/den0128
[PSA-CRYPT]	IHI 0086	*PSA Certified Crypto API*. arm-software.github.io/psa-api/crypto
[PSA-STAT]	ARM IHI 0097	*PSA Certified Status code API*. arm-software.github.io/psa-api/status-code
[PSA-FF-M]	ARM DEN 0063	*Arm® Platform Security Architecture Firmware Framework*. pages.arm.com/psa-apis

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 3 Terms and abbreviations

Term	Meaning
Application Root of Trust (ARoT)	This is the security domain in which additional security services are implemented. See *Platform Security Model* [PSM].
ARoT	See *Application Root of Trust*.
IMPLEMENTATION DEFINED	Behavior that is not defined by the this specification, but is defined and documented by individual implementations. Firmware developers can choose to depend on IMPLEMENTATION DEFINED behavior, but must be aware that their code might not be portable to another implementation.
Non-secure Processing Environment (NSPE)	This is the security domain outside of the *Secure Processing Environment*. It is the Application domain, typically containing the application firmware and hardware.
NSPE	See *Non-secure Processing Environment*.
Platform Root of Trust (PRoT)	The overall trust anchor for the system. This ensures the platform is securely booted and configured, and establishes the secure environments required to protect security services. See *Platform Security Model* [PSM].
PRoT	See *Platform Root of Trust*.

continues on next page

Table 3 – continued from previous page

Term	Meaning
Root of Trust (RoT)	This is the minimal set of software, hardware and data that is implicitly trusted in the platform — there is no software or hardware at a deeper level that can verify that the Root of Trust is authentic and unmodified.
Root of Trust Service (RoT Service)	A set of related security operations that are provided by a *Root of Trust*.
RoT	See *Root of Trust*.
RoT Service	See *Root of Trust Service*.
Secure Partition	A processing context with protected runtime state within the *Secure Processing Environment*. A secure partition may implement one or more *RoT Service*s, accessible via well-defined interfaces.
Secure Partition Manager (SPM)	Part of the *Secure Processing Environment* that is responsible for allocating resources to *Secure Partition*s, managing the isolation and execution of software within partitions, and providing IPC between partitions.
Secure Processing Environment (SPE)	This is the security domain that includes the *Platform Root of Trust* and the *Application Root of Trust* domains.
SPE	See *Secure Processing Environment*.
SPM	See *Secure Partition Manager*.

Potential for change

The contents of this specification are stable for version 1.0.

The following may change in updates to the version 1.0 specification:

- Small optional feature additions.
- Clarifications.

Significant additions, or any changes that affect the compatibility of the interfaces defined in this specification will only be included in a new major or minor version of the specification.

Conventions

Typographical conventions

The typographical conventions are:

italic	Introduces special terminology, and denotes citations.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS
: Used for some common terms such as IMPLEMENTATION DEFINED.

  Used for a few terms that have specific technical meanings, and are included in the *Terms and abbreviations*.

Red text
: Indicates an open issue.

Blue text
: Indicates a link. This can be

  - A cross-reference to another location within the document
  - A URL, for example example.com

Numbers

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

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

Feedback

We welcome feedback on the PSA Certified API documentation.

If you have comments on the content of this book, visit github.com/arm-software/psa-api/issues to create a new issue at the PSA Certified API GitHub project. Give:

- The title (Secure Storage API).
- The number and issue (IHI 0087 1.0.2).
- The location in the document to which your comments apply.
- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 About Platform Security Architecture

This document is one of a set of resources provided by Arm that can help organizations develop products that meet the security requirements of PSA Certified on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software. You can read more about PSA Certified here at www.psacertified.org and find more Arm resources here at developer.arm.com/platform-security-resources.

1.2 About the Secure Storage API

The interface described in this document is a PSA Certified API, that provides key/value storage interfaces for use with device-protected storage. The Secure Storage API describes two interfaces for storage:

Internal Trusted Storage API	An interface for storage provided by the *Platform Root of Trust* (PRoT).
Protected Storage API	An interface for external protected storage.

The Internal Trusted Storage API must be implemented in the PRoT as described in the *Platform Security Model* [PSM] specification.

If there are no *Application Root of Trust* (ARoT) services that rely on it, the Protected Storage API can be implemented in the *NSPE*. Otherwise, the Protected Storage API must be implemented in an ARoT within the *SPE*.

You can find additional resources relating to the Secure Storage API here at arm-software.github.io/psa-api/storage, and find other PSA Certified APIs here at arm-software.github.io/psa-api.

2 Architecture

2.1 Use Cases and Rationale

Two use cases are addressed by Secure Storage API:

- Secure storage for device intimate data (Internal Trusted Storage).
- Protection for data-at-rest (Protected Storage).

Internal Trusted Storage aims at providing a place for devices to store their most intimate secrets, either to ensure data privacy or data integrity. For example, a device identity key requires confidentiality, whereas an authority public key is public data but requires integrity. Other critical values that are part of a *Root of Trust Service* — for example, secure time values, monotonic counter values, or firmware image hashes — will also need trusted storage.

Protected Storage is meant to protect larger data-sets against physical attacks. It aims to provide the ability for a firmware developer to store data onto external flash, with a promise of data-at-rest protection, including device-bound encryption, integrity, and replay protection. It should be possible to select the appropriate protection level — for example, encryption only, or integrity only, or both — depending on the threat model of the device and the nature of its deployment.

2.2 Technical Background

Modern embedded platforms have multiple types of storage, each with different security properties.

Most embedded microprocessors (MCU) have on-chip flash storage that can be made inaccessible except to software running on the MCU. If the storage is made inaccessible to software other than that of the *Platform Root of Trust* (PRoT), then it can be used to store key material, replay protection values, or other data critical to the secure operation of the device.

In addition, many platforms also have external storage that requires confidentiality, integrity, and replay protection from attackers with physical access to the device.

By providing consistent APIs for accessing storage, software in both the *NSPE* and *SPE* can be written in a platform-independent manner. This improves portability between platforms that implement the PSA Certified APIs.

2.3 The Protected Storage API

The Protected Storage 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.

If the Protected Storage API is implemented using external storage without hardware protection, the data must be stored using authenticated encryption, as well as replay-protection values stored using the Internal Trusted Storage API. If the external storage has hardware protection — for example, remote locations or tamper proof enclosures — the need for cryptographic protection will be different.

Secure Storage API provides flags, PSA_STORAGE_FLAG_NO_CONFIDENTIALITY and PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION, enabling the caller to request a lower level of protection.

- PSA_STORAGE_FLAG_NO_CONFIDENTIALITY requests integrity but not confidentiality. For example, this might be selected when storing other party's public keys. This flag does not affect replay protection.
- PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION requests confidentiality and integrity protection of the data as controlled by PSA_STORAGE_FLAG_NO_CONFIDENTIALITY, but does not require the implementation to store data that would detect replacement with a previously valid value. For all other data objects, the implementation must ensure that the version returned is the most recently stored version.

Implementation note

This is usually achieved by creating a hash table or tree of all the file tags and storing the root in Internal Trusted Storage. Some implementations may only store the root and recreate the tree at boot — in which case when it detects and error it cannot tell which file has been tampered with and must reject all attempts to read replay protected files.

The implementation is permitted to treat these flags as indicative, and to apply a higher level of protection if it does not implement every protection class. It must not apply a lower level of protection than that requested.

An implementation must treat the PSA_STORAGE_FLAG_WRITE_ONCE flag as definitive, if it is supported.

When reporting meta data, psa_ps_get_info() should report the actual protection level applied, and not the requested level.

2.4 The Internal Trusted Storage API

The Internal Trusted Storage API is a more specialized API. Uses of the Internal Trusted Storage 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 used by components of the *PRoT*.

Storing assets that don't fit this requirement is permissible. In fact, it is expected that many platforms will have the Protected Storage API call directly into the Internal Trusted Storage API. For example, this can be done on platforms that do not have external flash.

While this document makes no requirements about the size of the storage available by the Internal Trusted Storage API, it is expected to be limited, and therefore should be used for small, security-critical values.

As the Internal Storage is implicitly confidential and protected from replay, the implementation can ignore the flags requesting lower levels of protection. However, it must honor the PSA_STORAGE_FLAG_WRITE_ONCE flag.

2.5 UIDs

`uids` in the Secure Storage API are defined as `uint64_t`. This is expected to be larger than would be used on any system. This large namespace is chosen to allow a *Root of Trust Service* to easily manage assets on behalf of other services.

For example, consider a cryptography service running as a RoT Service. When a service running in a *Secure Partition* requests key storage from the cryptography service, the cryptography service can concatenate a numerical identity of the requesting partition (for example, a `int32_t` in the *Arm® Platform Security Architecture Firmware Framework* [PSA-FF-M]) with the key identifier (for example, a `uint32_t` in the *PSA Certified Crypto API* [PSA-CRYPT]) to generate the `uid` of the Internal Trusted Storage entry for the key. This allows the cryptography service to easily manage isolation between the key namespaces of its various clients.

Requirements for `uid`:

- The value zero (`0`) is reserved, and will result in an error if passed to any of the Secure Storage API functions.
- Each partition can use any of the non-zero `uids` in the full 64-bit range.
- `uid` namespaces are independent. Using a `uid` in one partition has no impact on the `uids` or data assets in another partition.
- Data assets are always private. There is no mechanism that enables one partition to access a data asset owned by another partition.

The implication is that the implementation cannot divide the `uid` range between partitions, but it must use a partition identify, in addition to the `uid`, to identify a specific data asset.

2.6 Atomicity of Operations

In the event of power failure or other interruption of operations that modify storage, implementations of the Secure Storage API must maintain the properties shown in Table 4.

Table 4 Properties of storage operations

Atomicity	After the operation, the data assets of the storage service either contain the new data or are unchanged. Atomicity should be guaranteed in all situations — for example, an invalid request, a software crash or a power cycle — and must not result in corruption of the data assets. The only exceptions to this are situations involving storage failures or corruption.
Consistency	In the Secure Storage API, each operation is individually atomic. A multi-threaded application using Secure Storage API must not be able to observe any intermediate state in the data assets. If thread 'B' calls the Secure Storage API while thread 'A' is in the middle of an operation that modifies a data asset, thread 'B' must either see the state of the asset before, or the state of the asset after, the operation requested by thread 'A'.

continues on next page

Table 4 – continued from previous page

Isolation	A partition using the storage service cannot cause a change in the data assets belonging to a different partition.
Durability	When an operation that modifies storage returns to the caller, the data is persisted. System reset or power fail at this point will not revert the data assets to the previous state.

2.7 Components

Table 5 lists the significant components in a system that implements Secure Storage API.

Table 5 Components in a system that implements the Trusted Storage API

Component	Description
Internal Trusted Storage API	The storage API described in this document intended for access to internal flash memory.
Internal Trusted Storage service	A *Platform Root of Trust* service that implements the Internal Trusted Storage API.
Protected Storage API	The general-purpose storage API described in this document.
Protected Storage service	A service, implemented either in the *Application Root of Trust* or the *NSPE*, that implements the Protected Storage API.
Secure Partition Manager	The entity in the *Secure Processing Environment* responsible for communicating requests between the various secure services.

3 Requirements

3.1 Protected Storage requirements

1. The technology and techniques used by the Protected Storage service must allow for frequent writes and data updates.
2. If writing to external storage, the Protected Storage service must provide confidentiality — unless the caller specifically requests integrity only.
3. Confidentiality for a Protected Storage service may be provided by cryptographic ciphers using device-bound keys, a tamper resistant enclosure, or an inaccessible deployment location, depending on the threat model of the deployed system. If using counter-based encryption, the service must ensure a fresh key and nonce pair is used for each object instance encrypted.
4. If writing to external storage, the Protected Storage service must provide integrity protection.
5. Integrity protection for a Protected Storage service may be provided by cryptographic Message Authentication Codes (MAC) or signatures generated using device-bound keys, a tamper resistant enclosure, or an inaccessible deployment location, depending on the threat model of the deployed system.
6. If writing to external storage, the Protected Storage service must provide replay protection by writing replay protection values through the Internal Trusted Storage API, unless the caller specifically requests no replay protection.
7. If providing services to *Secure Partition*s, and the system isolates partitions from each other, then the Protected Storage service must provide protection from one partition accessing the storage assets of a different partition.
8. The Protected Storage service must use the partition identifier associated with each request for its access control mechanism.
9. If the Protected Storage service is providing services to other *ARoT* services, it must be implemented inside the ARoT itself.
10. If implemented inside the ARoT, the Protected Storage service can use helper services outside of the ARoT to perform actual read and write operations through the external interface or file system.
11. In the event of power failures or unexpected flash write failures, the implementation must attempt to fallback to allow retention of old content.
12. The creation of a `uid` with value `0` (zero) must be treated as an error.

3.2 Internal Trusted Storage requirements

1. The storage underlying the Internal Trusted Storage service must be protected from read and modification by attackers with physical access to the device.
2. The storage underlying the Internal Trusted Storage service must be protected from direct read or write access from software partitions outside of the *Platform Root of Trust*.
3. The technology and techniques used by the Internal Trusted Storage service must allow for frequent writes and data updates.
4. Confidentiality of data stored by the Internal Trusted Storage service can be implemented using an inaccessible deployment location, cryptographic ciphers, or a combination of these techniques.
5. Integrity of data stored by the Internal Trusted Storage service can be implemented using an inaccessible deployment location, cryptographic Message Authentication Codes (MAC) or signatures, or a combination of these techniques.
6. The Internal Trusted Storage service must provide protection from one partition accessing the storage assets of a different partition.
7. The Internal Trusted Storage service must use the partition identifier associated with each request for its access control mechanism.
8. The medium and methods utilized by a Internal Trusted Storage service must provide confidentiality within the threat model of the system.
9. The medium and methods utilized by a Internal Trusted service must provide integrity within the threat model of the system.
10. If the Debug Lifecycle state allows for a device to be debugged after deployment, then the Internal Trusted Storage service must provide confidentiality and integrity using cryptographic primitives with keys that are unavailable in the debug state.
11. If the device supports the `RECOVERABLE_PSA_ROT_DEBUG` Lifecycle state, then the Internal Trusted Storage service must provide confidentiality and integrity using cryptographic primitives with keys that are unavailable in the `RECOVERABLE_PSA_ROT_DEBUG` state.
12. In the event of power failures or unexpected flash write failures, the implementation must attempt to fallback to allow retention of old content.
13. In the extreme case of storage medium being completely non-accessible, no assurances can be made about the availability of the old content.
14. The `PSA_STORAGE_FLAG_WRITE_ONCE` must be enforced when the Root of Trust Lifecycle state of the device is `SECURED` or `NON_PSA_ROT_DEBUG`. It must not be enforced when the device is in the `PSA_ROT_PROVISIONING` state.
15. The creation of a `uid` with value `0` (zero) must be treated as an error.

The lifecycle states are described in *Platform Security Model* [PSM] and *Arm® Platform Security Architecture Firmware Framework* [PSA-FF-M].

4 Theory of Operation

4.1 Internal Trusted Storage API

The Internal Trusted Storage service that implements the Internal Trusted Storage API is not expected to replace the need for a filesystem that resides on external storage. Instead, it's intended to be used to interface to a small piece of storage that is only accessible to software that is part of the *Platform Root of Trust*. The Internal Trusted Storage API can be made accessible to the *Non-secure Processing Environment* as well as the *Secure Processing Environment*.

Internally the Internal Trusted Storage service should be designed such that one partition cannot access the data owned by another partition. The method of doing this is not specified here, but one method would be to store metadata with the data indicating the partition that owns it.

Figure 1 provides a simple example of how an Internal Trusted Storage service can be used by a service that implements *PSA Certified Crypto API* [PSA-CRYPT] to secure key-store material. This is illustrative and not prescriptive.

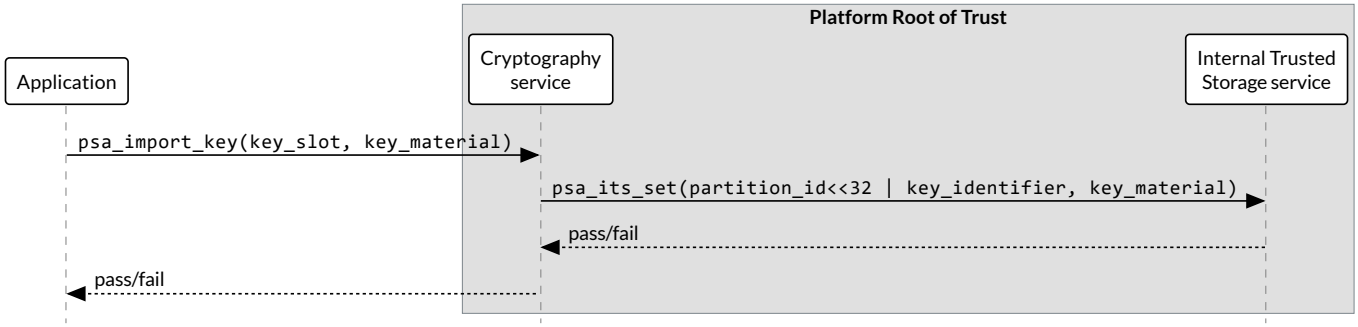

Figure 1 Sample Storage implementation with a service implementing the Crypto API

4.2 Memory access errors

When specifying an input or output buffer, the caller should ensure that the entire buffer is within memory it can access.

Attempting to reference memory that does not belong to the caller will either result in a memory access violation or will cause the function to return `PSA_ERROR_INVALID_ARGUMENT`.

Implementations of the Internal Trusted Storage API and Protected Storage API must check the length parameters of a buffer before attempting to access them. It is permissible to pass a null pointer to a zero length buffer.

5 API Reference

5.1 Status codes

The Secure Storage API uses the status code definitions that are shared with the other PSA Certified APIs.

The following elements are defined in `psa/error.h` from *PSA Certified Status code API* [PSA-STAT] (previously defined in [PSA-FF-M]):

```
typedef int32_t psa_status_t;

#define PSA_SUCCESS ((psa_status_t)0)

#define PSA_ERROR_GENERIC_ERROR         ((psa_status_t)-132)
#define PSA_ERROR_NOT_PERMITTED         ((psa_status_t)-133)
#define PSA_ERROR_NOT_SUPPORTED         ((psa_status_t)-134)
#define PSA_ERROR_INVALID_ARGUMENT      ((psa_status_t)-135)
#define PSA_ERROR_DOES_NOT_EXIST        ((psa_status_t)-140)
#define PSA_ERROR_INSUFFICIENT_STORAGE  ((psa_status_t)-142)
#define PSA_ERROR_STORAGE_FAILURE       ((psa_status_t)-146)
#define PSA_ERROR_INVALID_SIGNATURE     ((psa_status_t)-149)
#define PSA_ERROR_DATA_CORRUPT          ((psa_status_t)-152)
```

These definitions must be available to an application that includes either of the `psa/internal_trusted_storage.h` or `psa/protected_storage.h` header files.

> **Implementation note**
>
> An implementation is permitted to define the status code interface elements within the Secure Storage API header files, or to define them via inclusion of a `psa/error.h` header file that is shared with the implementation of other PSA Certified APIs.

5.2 General Definitions

These definitions must be defined in the header file `psa/storage_common.h`.

5.2.1 `psa_storage_info_t` (struct)

A container for metadata associated with a specific `uid`.

```
struct psa_storage_info_t {
    size_t capacity;
    size_t size;
    psa_storage_create_flags_t flags;
};
```

Fields

`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 create

5.2.2 psa_storage_create_flags_t (typedef)

Flags used when creating a data entry.

```
typedef uint32_t psa_storage_create_flags_t;
```

5.2.3 psa_storage_uid_t (typedef)

A type for `uid` used for identifying data.

```
typedef uint64_t psa_storage_uid_t;
```

5.2.4 PSA_STORAGE_FLAG_NONE (macro)

```
#define PSA_STORAGE_FLAG_NONE 0u
```

No flags to pass.

5.2.5 PSA_STORAGE_FLAG_WRITE_ONCE (macro)

```
#define PSA_STORAGE_FLAG_WRITE_ONCE (1u << 0)
```

The data associated with the `uid` will not be able to be modified or deleted. Intended to be used to set bits in `psa_storage_create_flags_t`.

5.2.6 PSA_STORAGE_FLAG_NO_CONFIDENTIALITY (macro)

```
#define PSA_STORAGE_FLAG_NO_CONFIDENTIALITY (1u << 1)
```

The data associated with the `uid` is public and therefore does not require confidentiality. It therefore only needs to be integrity protected.

5.2.7 PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION (macro)

```
#define PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION (1u << 2)
```

The data associated with the `uid` does not require replay protection. This can permit faster storage — but it permits an attacker with physical access to revert to an earlier version of the data.

5.2.8 PSA_STORAGE_SUPPORT_SET_EXTENDED (macro)

```
#define PSA_STORAGE_SUPPORT_SET_EXTENDED (1u << 0)
```

Flag indicating that psa_ps_create() and psa_ps_set_extended() are supported.

5.3 Internal Trusted Storage API

These definitions must be defined in the header file psa/internal_trusted_storage.h.

5.3.1 PSA_ITS_API_VERSION_MAJOR (macro)

The major version number of the Internal Trusted Storage API.

```
#define PSA_ITS_API_VERSION_MAJOR 1
```

It will be incremented on significant updates that can include breaking changes.

5.3.2 PSA_ITS_API_VERSION_MINOR (macro)

The minor version number of the Internal Trusted Storage API.

```
#define PSA_ITS_API_VERSION_MINOR 0
```

It will be incremented in small updates that are unlikely to include breaking changes.

5.3.3 psa_its_set (function)

Set the data associated with the specified uid.

```
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);
```

Parameters

uid	The identifier for the data.
data_length	The size in bytes of the data in p_data. If data_length == 0 the implementation will create a zero-length asset associated with the uid. While no data can be stored in such an asset, a call to psa_its_get_info() will return PSA_SUCCESS.
p_data	A buffer of data_length containing the data to store.
create_flags	The flags that the data will be stored with.

Returns: psa_status_t

A status indicating the success or failure of the operation.

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 either: • the uid is 0. • caller cannot access some or all of the memory in the range [p_data, p_data + data_length - 1].
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).

Description

Stores data in the internal storage.

- The uid value must not be zero.
- If uid exists it must not have been created as with PSA_STORAGE_FLAG_WRITE_ONCE — would result in PSA_ERROR_NOT_PERMITTED
- The caller must have access all memory from p_data to p_data + data_length.
- Even if all parameters are correct, the function can fail if there is insufficient storage space or in the case of a storage failure.

5.3.4 psa_its_get (function)

Retrieve data associated with a provided uid.

```
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);
```

Parameters

uid	The uid value.
data_offset	The starting offset of the data requested.
data_size	The amount of data requested.
p_data	On success, the buffer where the data will be placed.
p_data_length	On success, this will contain size of the data placed in p_data.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`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_INVALID_ARGUMENT`	The operation failed because either: • The `uid` is `0`. • The caller cannot access some or all of the memory in the range [p_data, `p_data + data_size - 1`]. • `data_offset` is larger than the size of the data associated with `uid`.
`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).

Description

Retrieves data associated with `uid`, starting at `data_offset` bytes from the beginning of the data. Fetches the lesser of `data_size` or `uid.size - data_offset` bytes, which can be zero.

`psa_its_get()` must not return bytes from beyond the end of `uid`.

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`. Any bytes beyond `p_data_length` are left unmodified.

If `data_size` is `0` or `data_offset == uid.size`, the contents of `p_data_length` will be set to zero, but the contents of `p_data` are unchanged. The function returns `PSA_SUCCESS`.

- The `uid` value must not be zero.
- The value of `data_offset` must be less than or equal to the length of `uid`.
- If `data_ffset` is greater than `uid.size`, no data is retrieved and the functions returns PSA_INVALID_ARGUMENT.
- If `data_size` is not zero, `p_data` must mot be `NULL`.
- The call must have access to the memory from `p_data` to `p_data + data_size - 1`.
- If the location `uid` exists the lesser of `data_size` or `uid.size - data_offset` bytes are written to the output buffer and `p_data_length` is set to the number of bytes written, which can be zero.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.3.5 psa_its_get_info (function)

Retrieve the metadata about the provided uid.

```
psa_status_t psa_its_get_info(psa_storage_uid_t uid,
                              struct psa_storage_info_t * p_info);
```

Parameters

uid	The uid value.
p_info	A pointer to the psa_storage_info_t struct that will be populated with the metadata.

Returns: psa_status_t

A status indicating the success or failure of the operation.

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_INVALID_ARGUMENT	The operation failed because either: • The uid is 0. • The caller cannot access some or all of the memory in the range [p_info, p_info + sizeof(psa_storage_info_t) - 1]
PSA_ERROR_STORAGE_FAILURE	The operation failed because the physical storage has failed (Fatal error).

Description

Retrieves the metadata stored for a given uid as a psa_storage_info_t structure.

- The uid value must not be zero.
- The call must have access to the memory from p_info to p_info + sizeof(psa_storage_info_t) - 1.
- If the location uid exists the metadata for the object is written to p_info.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.3.6 psa_its_remove (function)

Remove the provided uid and its associated data from the storage.

```
psa_status_t psa_its_remove(psa_storage_uid_t uid);
```

Parameters

`uid`	The `uid` value.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`PSA_SUCCESS`	The operation completed successfully.
`PSA_ERROR_NOT_PERMITTED`	The operation failed because the provided `uid` value was created with `PSA_STORAGE_FLAG_WRITE_ONCE`.
`PSA_ERROR_DOES_NOT_EXIST`	The operation failed because the provided `uid` value was not found in the storage.
`PSA_ERROR_INVALID_ARGUMENT`	The `uid` is `0`.
`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).

Description

Deletes the data from internal storage.

- The `uid` value must not be zero.
- If `uid` exists it and any metadata are removed from storage.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.4 Protected Storage API

These definitions must be defined in the header file `psa/protected_storage.h`.

5.4.1 `PSA_PS_API_VERSION_MAJOR` (macro)

The major version number of the Protected Storage API.

```
#define PSA_PS_API_VERSION_MAJOR 1
```

It will be incremented on significant updates that can include breaking changes.

5.4.2 `PSA_PS_API_VERSION_MINOR` (macro)

The minor version number of the Protected Storage API.

```
#define PSA_PS_API_VERSION_MINOR 0
```

It will be incremented in small updates that are unlikely to include breaking changes.

5.4.3 psa_ps_set (function)

Set the data associated with the specified `uid`.

```
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);
```

Parameters

`uid`	The identifier for the data.
`data_length`	The size in bytes of the data in `p_data`. If `data_length == 0` the implementation will create a zero-length asset associated with the `uid`. While no data can be stored in such an asset, a call to `psa_ps_get_info()` will return `PSA_SUCCESS`.
`p_data`	A buffer containing the data.
`create_flags`	The flags indicating the properties of the data.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`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 either: • The `uid` is `0`. • The operation failed because caller cannot access some or all of the memory in the range [`p_data`, `p_data + data_length - 1`].
`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.

Description

The newly created asset has a capacity and size that are equal to `data_length`.

- The `uid` value must not be zero.
- If `uid` exists it must not have been created as with `PSA_STORAGE_FLAG_WRITE_ONCE` - would result in `PSA_ERROR_NOT_PERMITTED`
- The caller must have access all memory from `p_data` to `p_data + data_length`.
- Even if all parameters are correct, the function can fail if there is insufficient storage space or in the case of a storage failure.

5.4.4 psa_ps_get (function)

Retrieve data associated with a provided `uid`.

```
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);
```

Parameters

`uid`	The `uid` value.
`data_offset`	The starting offset of the data requested. This must be less than or equal to `uid.size`.
`data_size`	The amount of data requested.
`p_data`	On success, the buffer where the data will be placed.
`p_data_length`	On success, will contain size of the data placed in `p_data`.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`PSA_SUCCESS`	The operation completed successfully.
`PSA_ERROR_INVALID_SIGNATURE`	The operation failed because the data associated with the `uid` failed authentication.
`PSA_ERROR_DOES_NOT_EXIST`	The operation failed because the provided `uid` value was not found in the storage.
`PSA_ERROR_INVALID_ARGUMENT`	The operation failed because either: • The `uid` is `0`. • The caller cannot access some or all of the memory in the range [`p_data`, `p_data + data_size - 1`]. • `data_offset` is larger than the size of the data associated with `uid`.
`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).
`PSA_ERROR_DATA_CORRUPT`	The operation failed because the data associated with the `uid` has been corrupted.
`PSA_ERROR_GENERIC_ERROR`	The operation failed because of an unspecified internal failure.

Description

Retrieves data associated with `uid`, starting at `data_offset` bytes from the beginning of the data. Fetches the smaller of `data_size` or `uid.size - data_offset` bytes. This can be zero.

`psa_ps_get()` must not return bytes from beyond the end of `uid`.

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`. Any bytes beyond `p_data_length` are left unmodified.

If `data_size` is `0` or `data_offset == uid.size`, the contents of `p_data_length` will be set to zero, but the contents of `p_data` are unchanged. The function returns `PSA_SUCCESS`.

- The `uid` value must not be zero.
- The value of `data_offset` must be less than or equal to the length of `uid`.
- If `data_offset` is greater than `uid.size` the function retrieves no data and returns `PSA_ERROR_INVALID_ARGUMENT`
- If `data_size` is not zero, `p_data` must mot be `NULL`.
- The call must have access to the memory from `p_data` to `p_data + data_size - 1`.
- If the location `uid` exists the lesser of `data_size` and `uid.size - data_offset` bytes are written to the output buffer and `p_data_length` is set to the number of bytes written, which can be zero.
- Any bytes in the buffer beyond `p_data_length` are left unmodified.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.4.5 `psa_ps_get_info` (function)

Retrieve the metadata about the provided `uid`.

```
psa_status_t psa_ps_get_info(psa_storage_uid_t uid,
                             struct psa_storage_info_t * p_info);
```

Parameters

`uid`	The identifier for the data.
`p_info`	A pointer to the `psa_storage_info_t` struct that will be populated with the metadata.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`PSA_SUCCESS`	The operation completed successfully.
`PSA_ERROR_INVALID_SIGNATURE`	The operation failed because the data associated with the `uid` failed authentication.
`PSA_ERROR_DOES_NOT_EXIST`	The operation failed because the provided `uid` value was not found in the storage.
`PSA_ERROR_INVALID_ARGUMENT`	The operation failed because either:

- The `uid` is `0`.
- The caller cannot access some or all of the memory in the range [`p_info`, `p_info + sizeof(psa_storage_info_t) - 1`]

`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).
`PSA_ERROR_DATA_CORRUPT`	The operation failed because the data associated with the `uid` has been corrupted.
`PSA_ERROR_GENERIC_ERROR`	The operation failed because of an unspecified internal failure.

Description

Retrieves the metadata stored for a given `uid` as a `psa_storage_info_t` structure.

- The `uid` value must not be zero.
- The call must have access to the memory from `p_info` to `p_info + sizeof(psa_storage_info_t) - 1`.
- If the location `uid` exists the metadata for the object is written to `p_info`.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.4.6 `psa_ps_remove` (function)

Remove the provided `uid` and its associated data from the storage.

```
psa_status_t psa_ps_remove(psa_storage_uid_t uid);
```

Parameters

`uid`	The identifier for the data to be removed.

Returns: `psa_status_t`

A status indicating the success or failure of the operation.

`PSA_SUCCESS`	The operation completed successfully.
`PSA_ERROR_NOT_PERMITTED`	The operation failed because the provided `uid` value was created with `PSA_STORAGE_FLAG_WRITE_ONCE`.
`PSA_ERROR_DOES_NOT_EXIST`	The operation failed because the provided `uid` value was not found in the storage.
`PSA_ERROR_INVALID_ARGUMENT`	The `uid` is `0`.
`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.

Description

Removes previously stored data and any associated metadata, including rollback protection data.

- The `uid` value must not be zero.
- If the location `uid` exists, it and any metadata are removed.
- Even if all parameters are correct, the function can fail in the case of a storage failure.

5.4.7 psa_ps_create (function)

Reserves storage for the specified `uid`.

```
psa_status_t psa_ps_create(psa_storage_uid_t uid,
                           size_t capacity,
                           psa_storage_create_flags_t create_flags);
```

Parameters

`uid`	A unique identifier for the asset.
`capacity`	The allocated capacity, in bytes, of the `uid`.
`create_flags`	Flags indicating properties of the storage.

Returns: `psa_status_t`

`PSA_SUCCESS`	The storage was successfully reserved.
`PSA_ERROR_INVALID_ARGUMENT`	The operation failed because the `uid` is `0`.
`PSA_ERROR_NOT_SUPPORTED`	The function is not implemented or one or more `create_flags` are not supported.
`PSA_ERROR_INSUFFICIENT_STORAGE`	`capacity` is bigger than the current available space.
`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).
`PSA_ERROR_GENERIC_ERROR`	The operation has failed due to an unspecified error.
`PSA_ERROR_ALREADY_EXISTS`	Storage for the specified `uid` already exists.

Description

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 the `psa_ps_set()` function is needed, calls to this function are redundant.

This function cannot be used to replace or resize 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 optional. Consult the platform documentation to determine if it is implemented or perform a call to `psa_ps_get_support()`. This function must be implemented if `psa_ps_get_support()` returns `PSA_STORAGE_SUPPORT_SET_EXTENDED`.

- The `uid` value must not be zero.
- If `uid` must not exist.
- The flag `PSA_STORAGE_FLAG_WRITE_ONCE` must not be set.
- Even if all parameters are correct, the function can fail if there is insufficient storage space or in the case of a storage failure.

5.4.8 psa_ps_set_extended (function)

Overwrite part of the data of the specified `uid`.

```
psa_status_t psa_ps_set_extended(psa_storage_uid_t uid,
                                 size_t data_offset,
                                 size_t data_length,
                                 const void * p_data);
```

Parameters

`uid`	The unique identifier for the asset.
`data_offset`	Offset within the asset to start the write.
`data_length`	The size in bytes of the data in `p_data` to write.
`p_data`	Pointer to a buffer which contains the data to write.

Returns: `psa_status_t`

`PSA_SUCCESS`	The asset exists, the input parameters are correct and the data is correctly written in the physical storage.
`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`.
`PSA_ERROR_INVALID_SIGNATURE`	The operation failed because the existing data failed authentication (MAC check failed).
`PSA_ERROR_DOES_NOT_EXIST`	The specified `uid` was not found.
`PSA_ERROR_INVALID_ARGUMENT`	The operation failed because either: • The `uid` is `0`. • The caller cannot access some or all of the memory in the range [`p_data`, `p_data + data_size - 1`]. • One or more of the preconditions regarding `data_offset`, `size`, or `data_length` was violated.
`PSA_ERROR_NOT_SUPPORTED`	The implementation does not support this function.
`PSA_ERROR_STORAGE_FAILURE`	The operation failed because the physical storage has failed (Fatal error).
`PSA_ERROR_DATA_CORRUPT`	The operation failed because the existing data has been corrupted.
`PSA_ERROR_GENERIC_ERROR`	The operation failed due to an unspecified error.

Description

Sets partial data into an asset based on the given identifier, `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.

This function is optional. Consult the platform documentation to determine if it is implemented or perform a call to `psa_ps_get_support()`. This function must be implemented if `psa_ps_get_support()` returns `PSA_STORAGE_SUPPORT_SET_EXTENDED`.

- The `uid` value must not be zero.
- If `uid` exists it must not have been created as with `PSA_STORAGE_FLAG_WRITE_ONCE` - would result in `PSA_ERROR_NOT_PERMITTED`
- `data_offset <= size`
- `data_offset + data_length <= capacity`
- Even if all parameters are correct, the function can fail in the case of a storage failure.

On Success:

- `size = max(size, data_offset + data_length)`
- `capacity` unchanged.

5.4.9 `psa_ps_get_support` (function)

Returns a bitmask with flags set for the optional features supported by the implementation.

```
uint32_t psa_ps_get_support(void);
```

Returns: `uint32_t`

Description

Currently defined flags are limited to:

- `PSA_STORAGE_SUPPORT_SET_EXTENDED`

Appendix A: Example header files

Each implementation of the Secure Storage API must provide a header file named `psa/storage_common.h`, and also any of `psa/internal_trusted_storage.h` and `psa/protected_storage.h` for the APIs that are implemented.

This appendix provides examples of the header files with all of the API elements. This can be used as a starting point or reference for an implementation.

A.1 psa/storage_common.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Secure Storage API v1.0.1
 *
 * This file includes common definitions
 */

#ifndef PSA_STORAGE_COMMON_H
#define PSA_STORAGE_COMMON_H

#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

struct psa_storage_info_t {
    size_t capacity;
    size_t size;
    psa_storage_create_flags_t flags;
};
typedef uint32_t psa_storage_create_flags_t;
typedef uint64_t psa_storage_uid_t;
#define PSA_STORAGE_FLAG_NONE 0u
#define PSA_STORAGE_FLAG_WRITE_ONCE (1u << 0)
#define PSA_STORAGE_FLAG_NO_CONFIDENTIALITY (1u << 1)
#define PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION (1u << 2)
#define PSA_STORAGE_SUPPORT_SET_EXTENDED (1u << 0)

#ifdef __cplusplus
}
#endif

#endif // PSA_STORAGE_COMMON_H
```

A.2 psa/internal_trusted_storage.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Secure Storage API v1.0.1
 *
 * This file describes the Internal Trusted Storage API
 */

#ifndef PSA_INTERNAL_TRUSTED_STORAGE_H
#define PSA_INTERNAL_TRUSTED_STORAGE_H

#include <stddef.h>
#include <stdint.h>

#include "psa/error.h"
#include "psa/storage_common.h"

#ifdef __cplusplus
extern "C" {
#endif

#define PSA_ITS_API_VERSION_MAJOR 1
#define PSA_ITS_API_VERSION_MINOR 0
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);
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);
psa_status_t psa_its_get_info(psa_storage_uid_t uid,
                              struct psa_storage_info_t * p_info);
psa_status_t psa_its_remove(psa_storage_uid_t uid);

#ifdef __cplusplus
}
#endif

#endif // PSA_INTERNAL_TRUSTED_STORAGE_H
```

A.3 psa/protected_storage.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Secure Storage API v1.0.1
 *
 * This file describes the Protected Storage API
 */

#ifndef PSA_PROTECTED_STORAGE_H
```

(continues on next page)

(continued from previous page)

```
#define PSA_PROTECTED_STORAGE_H

#include <stddef.h>
#include <stdint.h>

#include "psa/error.h"
#include "psa/storage_common.h"

#ifdef __cplusplus
extern "C" {
#endif

#define PSA_PS_API_VERSION_MAJOR 1
#define PSA_PS_API_VERSION_MINOR 0
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);
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);
psa_status_t psa_ps_get_info(psa_storage_uid_t uid,
                             struct psa_storage_info_t * p_info);
psa_status_t psa_ps_remove(psa_storage_uid_t uid);
psa_status_t psa_ps_create(psa_storage_uid_t uid,
                           size_t capacity,
                           psa_storage_create_flags_t create_flags);
psa_status_t psa_ps_set_extended(psa_storage_uid_t uid,
                                 size_t data_offset,
                                 size_t data_length,
                                 const void * p_data);
uint32_t psa_ps_get_support(void);

#ifdef __cplusplus
}
#endif

#endif // PSA_PROTECTED_STORAGE_H
```

Appendix B: Document history

Date	Release	Details
2019-02-25	*1.0 Beta 2*	First Release
2019-06-12	*1.0 Rel*	Final 1.0 API • The protected storage API now supports flags `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` and `PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION`. • Error values now use standard PSA error codes, which are now defined in `<psa/error.h>`. • Input parameters are now separate from output parameters. There are no longer any in/out parameters. • Size types have been replaced with `size_t` instead of `uint32_t`.
2022-10-17	*1.0.1 Rel*	• Relicensed the document under Attribution-ShareAlike 4.0 International with a patent license derived from Apache License 2.0. See *License* on page iv. • Documentation clarifications.
2023-03-23	*1.0.2 Rel*	• Clarified the protection requirements for ITS. See *Internal Trusted Storage requirements* on page 14. • Fixed inconsistent descriptions of `PSA_ERROR_STORAGE_FAILURE` errors.

Index of API elements