PSA Certified Secure Storage API 1.0

Document number:	IHI 0087
Release Quality:	Final
Issue Number:	3
Confidentiality:	Non-confidential
Date of Issue:	22/01/2024

Contents

About this document iv

Release information iv

License v

References vi

Terms and abbreviations vi

Potential for change vii

Conventions viii
Typographical conventions viii
Numbers viii

Feedback viii

1 Introduction 9

1.1 About Platform Security Architecture 9

1.2 About the Secure Storage API 9

2 Architecture 10

2.1 Use Cases and Rationale 10

2.2 Technical Background 10

2.3 The Protected Storage API 10

2.4 The Internal Trusted Storage API 11

2.5 UIDs 12

2.6 Atomicity of Operations 12

2.7 Components 13

3 Requirements 14

3.1 Protected Storage requirements 14

3.2 Internal Trusted Storage requirements 15

4 Theory of Operation 16

4.1 Internal Trusted Storage API 16

4.2 Memory access errors 16

5 API Reference 17

5.1 Status codes 17

5.2 General Definitions 17
5.2.1 `psa_storage_info_t` (struct) 18
5.2.2 `psa_storage_create_flags_t` (typedef) 18
5.2.3 `psa_storage_uid_t` (typedef) 18
5.2.4 `PSA_STORAGE_FLAG_NONE` (macro) 18
5.2.5 `PSA_STORAGE_FLAG_WRITE_ONCE` (macro) 18
5.2.6 `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` (macro) 19
5.2.7 `PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION` (macro) 19
5.2.8 `PSA_STORAGE_SUPPORT_SET_EXTENDED` (macro) 19

5.3 Internal Trusted Storage API 19
5.3.1 `PSA_ITS_API_VERSION_MAJOR` (macro) 19
5.3.2 `PSA_ITS_API_VERSION_MINOR` (macro) 19
5.3.3 `psa_its_set` (function) 20
5.3.4 `psa_its_get` (function) 21
5.3.5 `psa_its_get_info` (function) 22
5.3.6 `psa_its_remove` (function) 23

5.4 Protected Storage API 23
5.4.1 `PSA_PS_API_VERSION_MAJOR` (macro) 23
5.4.2 `PSA_PS_API_VERSION_MINOR` (macro) 24
5.4.3 `psa_ps_set` (function) 24
5.4.4 `psa_ps_get` (function) 25
5.4.5 `psa_ps_get_info` (function) 26
5.4.6 `psa_ps_remove` (function) 27
5.4.7 `psa_ps_create` (function) 28
5.4.8 `psa_ps_set_extended` (function) 29
5.4.9 `psa_ps_get_support` (function) 30

A Example header files 31

A.1 psa/storage_common.h 31

A.2 psa/internal_trusted_storage.h 32

A.3 psa/protected_storage.h 33

B Security Risk Assessment 35

B.1 About this assessment 35
B.1.1 Subject and scope 35
B.1.2 Risk assessment methodology 35

B.2 Feature definition 37
B.2.1 Introduction 37

B.2.2 Lifecycle 37
B.2.3 Operation and trust boundaries 38
B.2.4 Deployment models 38
B.2.5 Assumptions, constraints, and interacting entities 42
B.2.6 Stakeholders and Assets 42
B.2.7 Goals 42
B.2.8 Adversarial models 43

B.3 Threats 44
B.3.1 T.INTERFACE_ABUSE: Illegal inputs to the API 45
B.3.2 T.SPOOF_READ: Use the API to read another caller's data 45
B.3.3 T.SPOOF_WRITE: Use the API to modify another caller's data 46
B.3.4 T.EAVESDROPPING: Eavesdropping 47
B.3.5 T.MITM: Man In The Middle 48
B.3.6 T.DIRECT_READ: Bypassing the API to directly read data 49
B.3.7 T.DIRECT_WRITE: Bypassing the API to directly modify data 50
B.3.8 T.REPLACE: Physical replacement of the storage medium 51
B.3.9 T.GLITCH_READ: Glitching during a read 52
B.3.10 T.GLITCH_WRITE: Glitching during a write 52

B.4 Mitigation Summary 54
B.4.1 Architecture level mitigations 54
B.4.2 Implementation-level mitigations 54
B.4.3 User-level mitigations 56
B.4.4 Mitigations required by each deployment model 56

C Document history 58

Index of API elements 59

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.
January 2024	1.0.3	Non-confidential	Provide a Security Risk Assessment.

The detailed changes in each release are described in *Document history* on page 58.

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-FFM]	ARM DEN 0063	*Arm® Platform Security Architecture Firmware Framework*. developer.arm.com/documentation/den0063
[SP800-30]		NIST, *NIST Special Publication 800-30 Revision 1: Guide for Conducting Risk Assessments*, September 2012. doi.org/10.6028/NIST.SP.800-30r1

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*.

continues on next page

Table 3 – continued from previous page

Term	Meaning
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*.
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 1.0.3).
- 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-FFM]) 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-FFM].

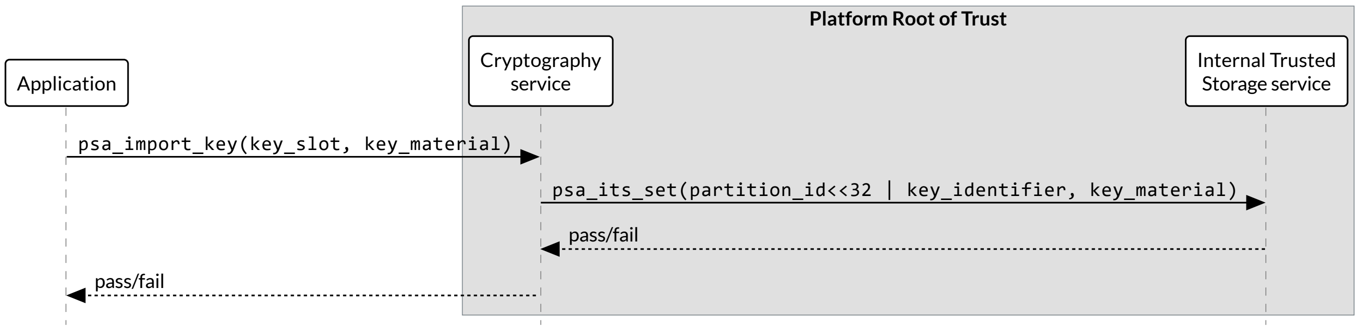
4 Theory of Operation

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.



Platform Root of Trust
Application
Cryptography service
Internal Trusted Storage service
psa_import_key(key_slot, key_material)
psa_its_set(partition_id<<32 | key_identifier, key_material)
pass/fail
pass/fail

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-FFM]):

```
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
```

(continues on next page)

(continued from previous page)

```
#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
#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
```

(continues on next page)

(continued from previous page)

```
#endif // PSA_PROTECTED_STORAGE_H
```

Appendix B: Security Risk Assessment

This appendix provides a Security Risk Assessment (SRA) of the Secure Storage API and of a generic implementation of storage. It describes the threats presented by various types of adversaries against the security goals for an implementation of a secure storage service, and mitigating actions for those threats.

- *About this assessment* describes the assessment methodology.
- *Feature definition* on page 37 defines the security problem.
- *Threats* on page 44 describes the threats and the recommended mitigating actions.
- *Mitigation Summary* on page 54 summarizes the mitigations, and where these are implemented.

B.1 About this assessment

B.1.1 Subject and scope

This SRA analyses the security of the Secure Storage API itself, and of the conceptual architectures for storage, not of any specific implementation of the API, or any specific use of the API. It does, however, divide implementations into four deployment models representing common implementation types, and looks at the different mitigations needed in each deployment model.

In this SRA:

- *Storage service* means the firmware implementing the Secure Storage API.
- *Storage medium* refers to the physical storage location.

B.1.2 Risk assessment methodology

Our risk ratings use an approach derived from *NIST Special Publication 800-30 Revision 1: Guide for Conducting Risk Assessments* [SP800-30]: for each Threat, we determine its Likelihood and the Impact. Each is evaluated on a 5-level scale, as defined in Table 6 and Table 7 on page 36.

Table 6 Likelihood levels

Level	Definition
Very Low	Unlikely to ever occur in practice, or *mathematically near impossible*
Low	The event could occur, but only if the attacker employs *significant* resources; or it is *mathematically unlikely*
Medium	A motivated, and well-equipped adversary can make it happen within the lifetime of a product based on the feature (resp. of the feature itself)
High	Likely to happen within the lifetime of the product or feature
Very High	Will happen, and soon (for instance a zero-day)

Table 7 Impact levels

Level	Definition	Example Effects
Very Low	Causes virtually no damage.	Probably none.
Low	The damage can easily be tolerated or absorbed.	There would be a CVE at most.
Medium	The damage will have a *noticeable* effect, such as *degrading* some functionality, but won't degrade completely the use of the considered functionality.	There would be a CVE at most.
High	The damage will have a *strong* effect, such as causing a significant reduction in its functionality or in its security guarantees.	Security Analysts would discuss this at length, there would be papers, blog entries. Partners would complain.
Very High	The damage will have *critical* consequences — it could kill the feature, by affecting several of its security guarantees.	It would be quite an event. Partners would complain strongly, and delay or cancel deployment of the feature.

For both Likelihood and Impact, when in doubt always choose the higher value. These two values are combined using Table 8 to determine the Overall Risk of a Threat.

Table 8 Overall risk calculation

	Impact				
Likelihood	**Very Low**	**Low**	**Medium**	**High**	**Very High**
Very Low	Very Low	Very Low	Very Low	Low	Low
Low	Very Low	Very Low	Low	Low	Medium
Medium	Very Low	Low	Medium	Medium	High
High	(Very) Low	Low	Medium	High	Very High
Very High	(Very) Low	Medium	High	Very High	Very High

Threats are handled starting from the most severe ones. Mitigations will be devised for these Threats one by one (note that a Mitigation may mitigate more Threats, and one Threat may require the deployment of more than one Mitigation to be addressed). Likelihood and Impact will be reassessed assuming that the Mitigations are in place, resulting in a Mitigated Likelihood (this is the value that usually decreases), a Mitigated Impact (it is less common that this value will decrease), and finally a Mitigated Risk. The Analysis is completed when all the Mitigated Risks are at the chosen residual level or lower, which usually is Low or Very Low.

The Mitigating actions that can be taken are defined in the acronym **CAST**:

- **Control**: Put in place steps to reduce the Likelihood and/or Impact of a Threat, thereby reducing the risk to an acceptable level.

- **Accept**: The threat is considered to be of acceptable risk such that a mitigation is not necessary or must be accepted because of other constraint or market needs.
- **Suppress**: Remove the feature or process that gives rise to the threat.
- **Transfer**: Identify a more capable or suitable party to address the risk and transfer the responsibility of providing a mitigation for the threat to them.

B.2 Feature definition

B.2.1 Introduction

Background

Introduction on page 9 provides the context in which the Secure Storage API is designed.

Purpose

The Secure Storage API separates the software responsible for providing the security of the data from the caller. The storage service calls on firmware that provides low level reads and writes of non-volatile storage medium and the access to any required bus. The Secure Storage API is to provide a consistent interface, so that applications do not need to account for the different low-level implementations.

This analysis does not address the engineering requirements to create a reliable storage medium from the underlying physical storage. It is assumed that the implementation will use the standard techniques, error correcting codes, wear levelling and so on, to ensure the storage is reliable.

B.2.2 Lifecycle

Figure 2 shows the typical lifecycle of a device.

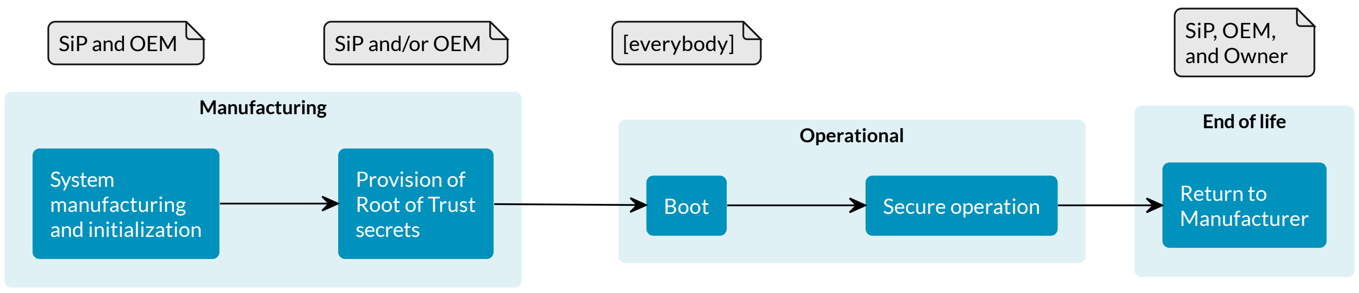


Figure 2 Device lifecycle of a system providing storage

The storage service, and the Secure Storage API are active during the operational phase, implemented within the boot-time and run-time software.

Within a boot session, it is the responsibility of the secure boot firmware to:

- Set up the isolation barriers between partitions.
- Provision the firmware implementing the storage service.
- Provision the credentials for authorizing the storage of data.
- Enable or disable debug facilities.

This SRA only considers threats to the storage service in its operational phase. The security of the boot process and of any data provisioning service are not considered in this SRA.

B.2.3 Operation and trust boundaries

Figure 3 shows all of the main components in the storage service. Presenting the context in which the Secure Storage API operates aids understanding of the threats and security mitigations and provides justification for some of the aspects of the API design.

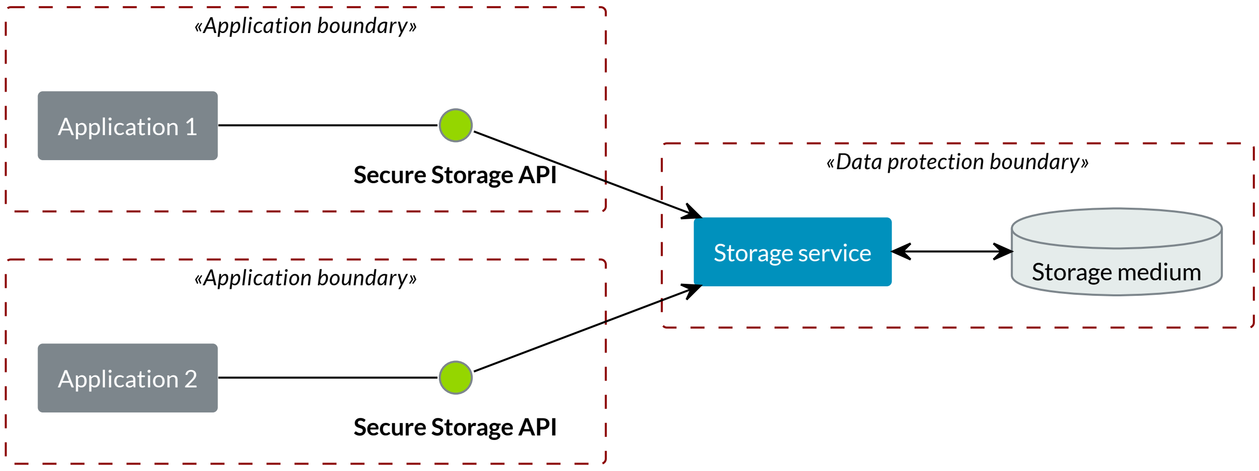


Figure 3 Trust boundaries of a system providing storage

Secure Storage API is a C language API. Therefore, any implementation of the API must execute, at least partially, within the context of the caller. When an implementation includes a trust boundary, the mechanism and protocol for communication across the boundary is not defined by this specification.

The operational dataflow diagram is reproduced for each of the deployment models. Although the dataflow itself is common to the models, the placement of trust boundaries is different.

It is helpful to visualize the effect of these differences on the threats against the dataflows.

B.2.4 Deployment models

DM.PROTECTED

The storage service and all physical storage is within the *Platform Root of Trust* (*PRoT*) partition. The *PRoT* partition has sole access to an area of non-volatile storage, thus that storage cannot be accessed by any other partition or any other means. This means that the storage service, any driver code, the storage service and storage medium all reside with the *PRoT* and are protected by the *PRoT*'s isolation mechanisms as shown in Figure 4 on page 39.

The storage service is the arbitrator of access from different applications and manages all data accesses (write, update and deletion). Therefore, the storage service is responsible for the SG.CONFIDENTIALITY, SG.INTEGRITY and SG.CURRENCY goals of each caller, including maintaining confidentiality between different callers.

An example of this deployment model is the use of on-chip flash or OTP with an access control mechanism such as a Memory Protection Unit.

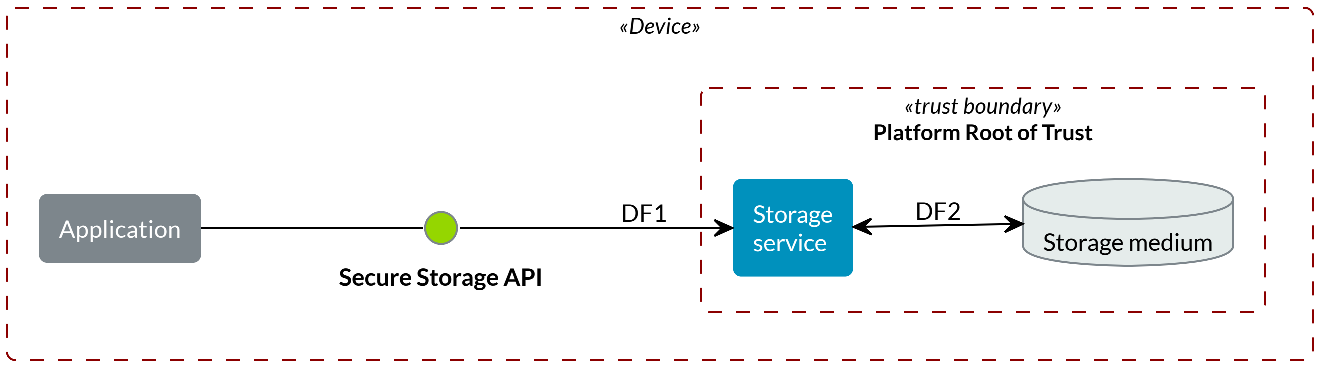


Figure 4 Trust boundaries in the deployment model DM.PROTECTED

DM.EXPOSED

The *PRoT* partition does not have sole access to the area of non-volatile storage, thus the storage medium can be read or written by another partition or by other means. This means that the driver code, or the storage medium resides outside the *PRoT* and is accessible to other partitions or by other means, as shown in as shown in Figure 5. Therefore, attackers can bypass the storage service.

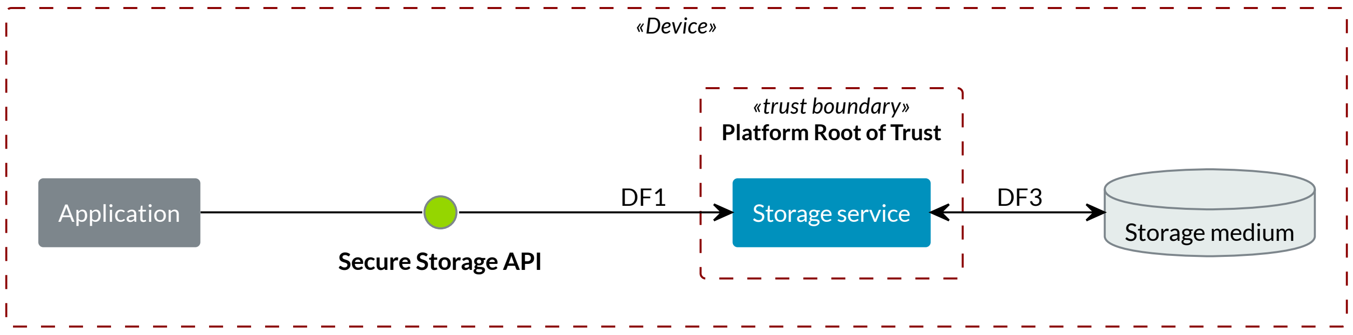


Figure 5 Trust boundaries in the deployment model DM.EXPOSED

The storage service is the arbitrator of access from different applications and manages accesses that write, update, and delete data. Therefore, the storage service is responsible for the SG.CONFIDENTIALITY, SG.INTEGRITY and SG.CURRENCY goal with respect to preventing access by a different caller.

The storage service cannot prevent other partitions or other means from reading or writing the storage, or accessing the link DF3. Therefore, the storage service is responsible for the SG.CONFIDENTIALITY, SG.INTEGRITY and SG.CURRENCY goals.

An example of this deployment model is the use of a file system on a flash chip.

DM.AUTHORIZED

There is a separate isolated storage medium that can only be accessed in response to an authenticated command and from which all replies include a means for verification of the response, as shown in Figure 6 on page 40. The isolation guarantees that there is no access to the storage medium other than by using the authentication mechanism.

The storage service is the arbitrator of access from different applications and manages those data accesses (write, update and deletion). Therefore, the storage service is responsible for the SG.CONFIDENTIALITY goal with respect to preventing access by a

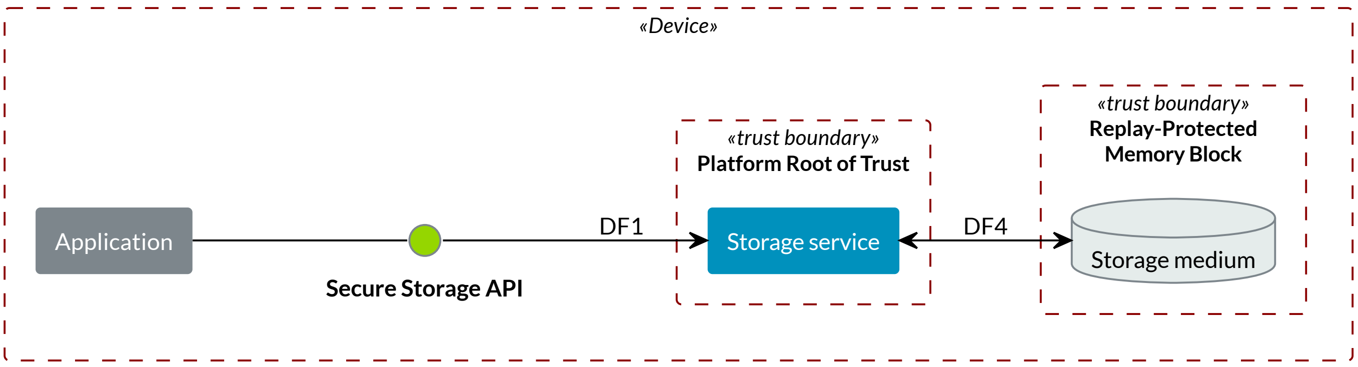


Figure 6 Trust boundaries in the deployment model DM.AUTHORIZED

different caller.

The authorization and verification mechanism provided by the storage medium controls access to data (reads, writes and modification). Therefore, the storage medium is responsible for the SG.INTEGRITY and SG.CURRENCY goals. Attacks on these mechanisms are out of scope.

However, the communication between the storage service and the storage medium is observable by other partitions and any other means as any data sent in plain text can be observed. Therefore, the storage service is responsible for SG.CONFIDENTIALITY.

The storage service and the storage medium are jointly responsible for protecting the assets required to authorize commands. Attacks on the storage service that expose these assets are in scope.

An example of this deployment model is the use of an RPMB memory block.

DM.SECURE_LINK

There is a separate isolated storage medium that can only be accessed across a cryptographically protected secure channel as shown in Figure 7. The secure channel protocol provides authentication, confidentiality and integrity of data in transit. The isolation guarantees that there is no access to the storage medium other than by using this channel.

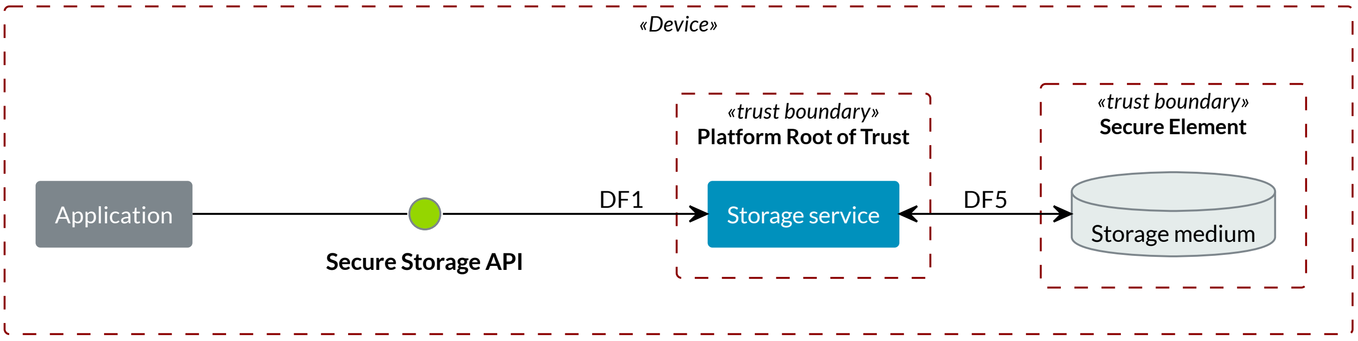


Figure 7 Trust boundaries in the deployment model DM.SECURE_LINK

The storage service is the arbitrator of access from different applications and manages those data accesses (write, update and deletion). Therefore, the storage service is responsible for the SG.CONFIDENTIALITY goal with respect to preventing access by a different caller.

The authorization and verification mechanism provided by the secure channel protocol controls access to data (reads, writes and modification). Therefore, the storage medium is responsible for the SG.INTEGRITY and SG.CURRENCY goals. Attacks on the storage medium are out of scope.

The communication between the storage service and the storage medium is protected from observation by other partitions and other means as the data is sent in encrypted form over the secure channel. Attacks on the secure channel protocol are out of scope.

The storage service uses the secure channel protocol, the storage service and the storage medium are jointly responsible for protecting the assets required to set up the channel. Attacks on the storage service that expose these assets are in scope.

An example of this deployment model is the use of a Secure Element, or a secure flash device.

Optional isolation

Implementations can isolate the storage service from the caller and can further isolate multiple calling applications. Various technologies can provide protection, for example:

- Process isolation in an operating system.
- Partition isolation, either with a virtual machine or a partition manager.
- Physical separation between execution environments.

The mechanism for identifying callers is beyond the scope of this specification. An implementation that provides caller isolation must document the identification mechanism. An implementation that provides caller isolation must document any implementation-specific extension of the API that enables callers to share data in any form.

In summary, there are three types of implementation:

- No isolation: there is no security boundary between the caller and the storage service. For example, a statically or dynamically linked library is an implementation with no isolation. As the caller is in the same security domain as the storage, the API cannot prevent access to the storage medium that does not go through the API.
- Simple Isolation: A single security boundary separates the storage service from the callers, but there is no isolation between callers. The only access to stored data is via the storage service, but the storage service cannot partition data between different callers.
- Caller isolation: there are multiple caller instances, with a security boundary between the caller instances among themselves, as well as between the storage service and the caller instances. For example, a storage service in a multiprocessor environment is an implementation with caller isolation. The only access to the stored data is via the storage service and the storage service can partition stored data between the different callers.

B.2.5 Assumptions, constraints, and interacting entities

This SRA makes the following assumptions about the Secure Storage API design:

- The API does not provide arguments that identify the caller, because they can be spoofed easily, and cannot be relied upon. It is assumed that the implementation of the API can determine the caller identity, where this is required. See *Optional isolation* on page 41.
- The API does not prevent the use of mitigations that are required by an implementation of the API. See *Mitigations that are transferred to the implementation* on page 54.
- The *Platform Security Model* [PSM] assumes that at least the code in the *Root of Trust* partitions (*PRoT* and *ARoT*) are verified at boot, and on any update. Therefore, it is assumed that this code is trustworthy. If any malicious code can run in the RoT partitions, it has achieved full control.
- For the purposes of this analysis, it is assumed that in deployment models DM.AUTHORIZED and DM.SECURE_LINK, there is no way to access the stored data without going through the authenticated channel. That is, an attack that would expose the physical storage medium is beyond the resources of the attacker.
- The analysis ignores attacks that only result in a denial of service. There are many ways an attacker can deny service to the complete system, with or without involving the storage service.
- The analysis only looks at an active attack. However, data is also subject to accidental modification, for example from cosmic radiation causing a bit flip. Therefore, standard engineering practice — such as use of error correcting codes — should be taken to protect data.

B.2.6 Stakeholders and Assets

This analysis looks at the security from the point of view of the applications that call on the service to store data, and on the overall system.

The following assets are considered in this assessment:

Data to be stored
: The purpose of a storage service is to securely store data for its callers.

Caller Identities
: To ensure that data stored for one caller is not revealed to a different caller, each caller must have a unique identity.

Implementation Secrets
: If in order to secure the data, the storage service uses encryption keys for confidentiality and integrity, these mut be considered assets of the storage service.

B.2.7 Goals

SG.CONFIDENTIALITY
: An adversary is unable to disclose Stored Data that belongs to a different Stored Data Owner. A legitimate owner can guarantee their data has not been exposed.

SG.INTEGRITY
: An adversary is unable to modify Stored Data that belongs to a different Stored Data Owner, to a value that was not previously stored by the Stored Data Owner. A legitimate owner can guarantee that data returned is a value they have stored.

SG.CURRENCY

An adversary is unable to modify Stored Data that belongs to a different Stored Data Owner. The legitimate owner can guarantee that data returned is the most recent value that have stored.

B.2.8 Adversarial models

Adversarial models are descriptions of capabilities that adversaries of systems implementing the Secure Storage API can have, grouped into classes. The adversaries are defined in this way to assist with threat modelling an abstract API, which can have different implementations, in systems with a wide range of security sensitivity.

AM.0 The Adversary is only capable of accessing data that requires neither physical access to a system containing an implementation of the feature nor the ability to run software on it. This Adversary is intercepting or providing data or requests to the target system via a network or other remote connection.

For instance, the Adversary can:

- Read any input and output to the target through external apparatus.
- Provide, forge, replay or modify such inputs and outputs.
- Perform timings on the observable operations being done by the target, either in normal operation or as a response to crafted inputs. For example, timing attacks on web servers.

AM.1 The Adversary can additionally mount attacks from software running on a target processor implementing the feature. This type of Adversary can run software on the target.

For instance, the Adversary can:

- Attempt software exploitation by running software on the target.
- Exploit access to any memory mapped configuration, monitoring, debug register.
- Mount any side channel analysis that relying on software-exposed built-in hardware features to perform physical unit and time measurements.
- Perform software-induced glitching of resources such as Rowhammer, RASpberry or crashing the CPU by running intensive tasks.

AM.2 In addition to the above, the Adversary is capable of mounting hardware attacks and fault injection that does not require breaching the physical envelope of the chips. This type of Adversary has access to a system containing an implementation of the target feature.

For instance, the Adversary can:

- Conduct side-channel analysis that requires measurement equipment. For example, this can utilize leakage sources such as EM emissions, power consumption, photonics emission, or acoustic channels.
- Plug malicious hardware into an unmodified system.
- Gain access to the internals of the target system and interpose the SoC or memory for the purposes of reading, blocking, replaying, and injecting transactions.
- Replace or add chips on the motherboard.
- Make simple, reversible modifications, to perform glitching.

AM.3 In addition to all the above, the Adversary can perform invasive SoC attacks.

For instance, the Adversary can:

- Decapsulate a chip, via laser or chemical etching, followed by microphotography to reverse engineer the chip.
- Use a focused ion beam microscope to perform gate level modification.

The adversarial models that are in scope depend on the product requirements. To ensure that the Secure Storage API can be used in a wide range of systems, this assessment considers adversarial models AM.0, AM.1, and AM.2 to be in-scope.

Code in the RoT partitions is assumed to be trustworthy — and any untrustworthy code running in *PRoT* partitions already has complete control of the target — therefore, in AM.1 this SRA only considers threats from malicious actors running in *Non-secure Processing Environment*.

B.3 Threats

Because Secure Storage API can be used in a wide range of deployment models and a wide range of threats, not all mitigating actions apply to all deployment models. As a result, various mitigations are optional to implement, depending on which threats exist in a particular domain of application, and which deployment model is used.

Table 9 summarizes the threats.

Table 9 Summary of threats

Threat	Description
T.INTERFACE_ABUSE	Call the API with illegal inputs
T.SPOOF_READ	Reading data for a different caller using the API
T.SPOOF_WRITE	Writing data for a different caller using the API
T.EAVESDROPPING	Accessing data in transit
T.MITM	A Man in the Middle can actively interfere with communication
T.DIRECT_READ	Directly reading stored data, bypassing the API
T.DIRECT_WRITE	Directly modifying data, bypassing the API
T.REPLACE	Physical replacement of the storage medium
T.GLITCH_READ	Glitching during a read
T.GLITCH_WRITE	Glitching during a write

B.3.1 T.INTERFACE_ABUSE: Illegal inputs to the API

Description: An attacker can abuse the Secure Storage API. For example:

- Passing out of range values to the interface to provoke unexpected behavior of the implementation.
- Passing invalid input or output buffers to the interface, that would cause the implementation to access non-existent memory, or memory that is inaccessible to the caller — including accessing assets of the storage service.

Adversarial Model	AM.1
Security Goal	SG.CONFIDENTIALITY, SG.INTEGRITY
Unmitigated Impact	Very High
Unmitigated Likelihood	Very High
Unmitigated Risk	Very High
Mitigating Actions	**M.ValidateParameter. Transfer** to the implementation: check all API parameters to lie within valid ranges, including memory access permissions. **M.MemoryBuffer. Control** by API design: input buffers are fully consumed by the implementation before returning from a function. An implementation must not access the caller's memory after a function has returned.
Residual Impact	Very High
Residual Likelihood	Very Low
Residual Risk	Low

B.3.2 T.SPOOF_READ: Use the API to read another caller's data

Description: In all deployment models, an attacker attempts to read data stored for another caller using the Secure Storage API.

The API does not require that the names used by caller for stored data are globally unique, only unique within that caller's namespace.

Adversarial Model	AM.1
Security Goal	SG.CONFIDENTIALITY
Unmitigated Impact	Very High
Unmitigated Likelihood	Very High
Unmitigated Risk	Very High
Mitigating Actions	**M.ImplicitIdentity. Control** by API design: caller identity is not provided by the caller to the API. If caller identity is supplied by the caller in the API, the identity can be spoofed by another caller. Using authentication credentials only moves the problem of storing secrets, but does not solve it.

	Transfer to the implementation: provide caller identities, to isolate data that belongs to different callers. The assurance that the storage service can give is limited by the assurance that the implementation can give as to the identity of the caller. Where each user runs in a separate partition, the identity can be provided by the partition manager. Where different users run within a single partition, **Transfer** the responsibility for separating users within that partition to the operating system or run time within that partition. **M.FullyQualifiedNames**. **Transfer** to the implementation: use a fully-qualified data identifier, that is a combination of an owner identity and the item UID. The implementation must used the owner identity to ensure that a data request to the storage service does not return data of the same UID, that was stored by a different caller. The storage service must also ensure that if a data item with the fully-qualified identifier does not exist, the implementation returns the correct error.
Residual Impact	Very High
Residual Likelihood	Very Low
Residual Risk	Low

B.3.3 T.SPOOF_WRITE: Use the API to modify another caller's data

Description: In all deployment models, an attacker attempts to write data to a file belonging to another caller using the Secure Storage API or create a new file in a different caller's namespace.

This threat is the counterpart to T.SPOOF_READ except that the attacker tries to write data rather than read. It is therefore subject to the same analysis.

Adversarial Model	AM.1
Security Goal	SG.CONFIDENTIALITY
Unmitigated Impact	Very High
Unmitigated Likelihood	Very High
Unmitigated Risk	Very High
Mitigating Actions	M.FullyQualifiedNames, M.ImplicitIdentity.
Residual Impact	Very High
Residual Likelihood	Very Low
Residual Risk	Low

B.3.4 T.EAVESDROPPING: Eavesdropping

Description: An attacker accesses data in transit, either between the caller and the storage service, or between the storage service and the storage medium.

In all deployment models, by the definition of an isolated partition in the *Platform Security Model* [PSM], transfer within the partition, and transfers between one *Secure Partition* and another are isolated from eavesdroppers. Therefore, if the caller is in a *Secure Partition*, there is no possibility of an eavesdropper accessing the data. However, if data is sent or returned to a caller in the *Non-secure Processing Environment* (NSPE), although the data is securely delivered to the *NSPE*, it is exposed to all users in the *NSPE*. As previously noted, the implementation **transfers** the duty of separating users in the *NSPE* to the OS.

For deployment model DM.PROTECTED, the storage service and the storage medium are isolated.

In DM.EXPOSED, any adversary that can obtain operating system privileges in the *NSPE* will have access to all the memory and will therefore be able to eavesdrop on all data in transit.

An attacker that is external to the processor, AM.2, will be able to exploit an eavesdropping attack if the bus to which the memory is attached is accessible via external pins. Otherwise, the attack is limited to internal attackers AM.1.

In DM.AUTHORIZED, an attacker with access to the bus, or to intermediate data buffers, can eavesdrop and obtain the messages.

In DM.SECURE_LINK, an attacker can only eavesdrop on any data transfer not protected by the secure channel.

Adversarial Model	AM.0, AM.1, AM.2			
Security Goal	SG.CONFIDENTIALITY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	Very High
Unmitigated Likelihood	n/a — except for transfer of data to clients in the *NSPE*	Very High	High	High
Unmitigated Risk	n/a	Very High	Very High	Very High
Mitigating Actions	**M.Encrypt. Transfer** to the implementation: for DM.EXPOSED and DM.AUTHORIZED, the data at rest must be encrypted. The storage service must apply the encryption to the data before it leaves the *PRoT* partition. The encryption mechanism chosen must be sufficiently robust. The key used for encryption must be sufficiently protected, that is, it must only be available to the storage service. **M.PRoTRootedSecLink. Transfer** to the implementation: for DM.SECURE_LINK, communication with the storage medium must be over a well-designed secure channel. If the secure channel is not rooted in the *PRoT* then any adversary in the partition (AM.1), or with access to the partition (AM.2), in which the channel terminates will be able to eavesdrop on traffic leaving the *PRoT* before it is encrypted. The secure channel must be rooted within the PRoT. However, the stored data does not need to be separately encrypted beyond the protection			

	provided by the secure channel. The private information required to establish the channel must be suitably protected by both the storage service and the storage medium. **M.UseSecurePartitions**. **Transfer** to the application: for all deployment models, place callers that handle sensitive data into separate partitions. To ensure that an attacker in the *NSPE* cannot access the data sent by the caller to the storage service, or the replies the storage service returns to the caller, place all code that needs to use the storage service into one or more *Secure Partition*, with one partition per service.			
Residual Impact	Very High	Very High	Very High	Very High
Residual Likelihood	n/a	Very Low	Very Low	Very Low
Residual Risk	n/a	Low	Low	Low

B.3.5 T.MITM: Man In The Middle

Description: An attacker can actively interfere with communication and replace the transmitted data. In this threat the SRA only considers attackers between the storage service and the storage medium. An attacker interposing between the Caller and the storage service is considered under T.SPOOF_READ or T.SPOOF_WRITE.

For DM.PROTECTED, the storage service and the storage medium are isolated.

For DM.EXPOSED, any code running in the *NSPE* has access to the storage medium and any driver firmware, and therefore can act as a man in the middle, by for example persuading the storage service to write to one buffer, and the storage medium to read from another.

For DM.AUTHORIZED, a man in the middle eavesdrops on data in transit.

For DM.SECURE_LINK, a naive secure channel is vulnerable to a man in the middle attack.

Adversarial Model	AM.1, AM.2			
Security Goal	SG.INTEGRITY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	High
Unmitigated Likelihood	n/a	Very High	High	High
Unmitigated Risk	n/a	Very High	Very High	High
Mitigating Actions	M.Encrypt. **Transfer** to the implementation: if data is encrypted, a man in the middle cannot know what data is being transferred. It also means they cannot force a specific value to be stored. **M.MAC**. **Transfer** to the implementation: for DM.EXPOSED, apply a Message Authentication Code or a signature to the stored data, or use an authenticated encryption scheme. If the storage service checks the MAC or tag when data is read back from the storage medium to detect unauthorized modification. **M.UniqueKeys**. **Transfer** to the implementation: for DM.AUTHORIZED and DM.SECURE_LINK, use unique keys for securing the authenticated or secure			

	channel. If the keys used by the storage medium are unique to each instance, as an attacker can only learn the key used on this specific instance. They cannot construct a class break by discovering the key for every instance. **M.VerifyReplies**. **Transfer** to the implementation: for DM.AUTHORIZED, commands and replies are authenticated by the storage medium. Therefore, the man in the middle cannot forge a valid reply which indicates that the data has been stored when it has not. If the storage service validates replies from the storage medium, it can verify that the data it sent was correctly stored, and the data retrieved is the value previously stored. **M.AuthenticateEndpoints**. **Transfer** to the implementation: for DM.SECURE_LINK, use mutual authentication of the storage service and storage medium when setting up the secure channel. For example, this can be achieved by using a single key, known only to both parties. **M.ReplayProtection**. **Transfer** to the implementation: for DM.AUTHORIZED and DM.SECURE_LINK, use replay protection in the communication protocol. This can be achieved by including a nonce in the construction of protocol messages. This enables the storage medium to detect attempts to replay previous commands and reject them.			
Residual Impact	Very High	Very High	High	High
Residual Likelihood	n/a	Very Low	Very Low	Very Low
Residual Risk	n/a	Low	Low	Low

B.3.6 T.DIRECT_READ: Bypassing the API to directly read data

Description: An attacker might be able to read stored data through a mechanism other than the API.

In DM.PROTECTED, no attacker should be able to access the stored data.

In DM.EXPOSED, all attackers can access the data.

In DM.AUTHORIZED, the attacker cannot form valid requests to access data. It can, however, eavesdrop on a legitimate request and replay it later.

In DM.SECURE_LINK, the attacker cannot form valid requests to access data. It can, however, eavesdrop on a legitimate request and even if it cannot understand it, it could replay it later.

Adversarial Model	AM.1, AM.2			
Security Goal	SG.CONFIDENTIALITY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	High
Unmitigated Likelihood	n/a	Very High	High	High
Unmitigated Risk	n/a	Very High	Very High	High

Mitigating Actions	M.ReplayProtection. **Transfer** to the implementation: for DM.AUTHORIZED and DM.SECURE_LINK, use replay protection in the communication protocol. M.Encrypt. **Transfer** to the implementation: for DM.EXPOSED and DM.AUTHORIZED, encrypting the data prevents disclosure.			
Residual Impact	Very High	Very High	High	High
Residual Likelihood	n/a	Very Low	Very Low	Very Low
Residual Risk	n/a	Low	Low	Low

B.3.7 T.DIRECT_WRITE: Bypassing the API to directly modify data

Description: An attacker might be able to modify data stored for another caller.

In DM.PROTECTED, no attacker should be able to access the stored data.

In DM.EXPOSED, the SRA assumes that any attacker capable of running code in the *NSPE* can modify the stored data. However, assuming it is encrypted, the attacker cannot create the correct ciphertext for chosen plain text.

In DM.AUTHORIZED, although the attacker cannot form a valid command, the attacker can eavesdrop on a legitimate request and replay it later.

In DM.SECURE_LINK, although the attacker cannot form a valid command, the attacker can eavesdrop on a legitimate request and replay it later.

Adversarial Model	AM.1 AM.2			
Security Goal	SG.INTEGRITY, SG.CURRENCY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	High
Unmitigated Likelihood	n/a	Very High	High	High
Unmitigated Risk	n/a	Very High	Very High	High
Mitigating Actions	M.Encrypt. **Transfer** to the implementation: encrypted data cannot be modified to an attacker-chosen plaintext value. However, an attacker can still corrupt the stored data. M.MAC. **Transfer** to the implementation: for DM.EXPOSED, integrity-protect the stored data using a MAC, signature, or AEAD scheme. The verification of data integrity must be implemented within the storage service in the *PRoT*, otherwise the result could be spoofed. M.ReplayProtection. **Transfer** to the implementation: for DM.AUTHORIZED and DM.SECURE_LINK, if the channel protocol includes replay protection, the storage medium will check the nonce for freshness, and prevent replay of old messages. **M.AntiRollback**. **Transfer** to the implementation: in DM.EXPOSED, M.MAC is insufficient to prevent an attacker from replacing one version of stored data — or the entire contents of the storage medium — with a previously stored version. The previously stored data would pass the integrity checks.			

To prevent this attack, the storage service must keep some authentication data in a location the attacker cannot access. This location could be stored within the *PRoT* partition, that is using the DM.PROTECTED, or in a separate secure enclave using the deployment model DM.AUTHORIZED or DM.SECURE_LINK. The data could be the root of a hash tree, or it could be a counter used with a root key to generate a version-specific MAC key.

In the case of a counter, some consideration should be given to the expected number of updates that will be made to the data. If the implementation only needs to offer rollback protection on firmware updates, where a low number is expected in the lifetime of the product and the counter could be stored in fuse. If the implementations needs to ensure the currency of a file store that is regularly updated — the number of updates could exhaust any practical number of fuses and would instead need a 32-bit counter.

Residual Impact	Very High	Very High	High	High
Residual Likelihood	n/a	Very Low	Very Low	Very Low
Residual Risk	n/a	Low	Low	Low

B.3.8 T.REPLACE: Physical replacement of the storage medium

Description: An attacker might physically replace the storage medium.

For DM.PROTECTED, it is not possible to replace the storage.

For DM.EXPOSED, if the storage medium is integrated with the chip, it is not possible to replace the storage. But in many systems, the storage medium will be on a separate device.

For DM.AUTHORIZED and DM.SECURE_LINK, it is possible to replace the storage medium.

Adversarial Model	AM.3			
Security Goal	SG.INTEGRITY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	Very High
Unmitigated Likelihood	n/a	Very High	High	High
Unmitigated Risk	n/a	Very High	Very High	Very High

Mitigating Actions

M.UniqueKeys and M.MAC. **Transfer** to the implementation: for DM.EXPOSED, use device-specific secret keys to authenticate the stored data. With unique authentication keys, data stored on one device cannot be verified on another device.

M.UniqueKeys and M.VerifyReplies. **Transfer** to the implementation: for DM.AUTHORIZED and DM.SECURE_LINK, use device-specific secret keys to authenticate the communication between the storage service and storage medium.

	In DM.AUTHORIZED, the attacker will not be able to find a new instance of the storage medium that can form the correct responses to commands. In DM.SECURE_LINK, the attacker will not be able to find a new instance of the storage medium that can complete the handshake to set up the secure channel.			
Residual Impact	Very High	Very High	High	High
Residual Likelihood	n/a	Very Low	Very Low	Very Low
Residual Risk	n/a	Low	Low	Low

B.3.9 T.GLITCH_READ: Glitching during a read

Description: An attacker with physical access might be able to disrupt the power or clock to cause a misread.

In this threat, an attacker with physical access to the device causes a power or frequency glitch to cause a misread. In particular, it might prevent the storage service from performing the verification of replies or causing it to ignore the result of any check. Thus, causing the storage service to return an incorrect value to the caller.

Adversarial Model	AM.3			
Security Goal	SG.INTEGRITY			
Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	Very High
Unmitigated Likelihood	High	High	Low	Low
Unmitigated Risk	Very High	Very High	Medium	Medium
Mitigating Actions	**M.GlitchDetection. Transfer** to the implementation: for all deployment models, active glitch detection circuits can raise an exception if a glitch is detected, permitting the computing circuitry to take corrective action.			
Residual Impact	Very High	Very High	Very High	Very High
Residual Likelihood	Low	Very Low	Very Low	Very Low
Residual Risk	Medium	Low	Low	Low

B.3.10 T.GLITCH_WRITE: Glitching during a write

Description: An attacker with physical access might be able to disrupt the power or clock to prevent a write from being completed.

In this threat, an attacker with physical access to the device causes a power or frequency glitch to cause a write to fail.

Adversarial Model	AM.3
Security Goal	SG.INTEGRITY

Deployment Model	DM.PROTECTED	DM.EXPOSED	DM.AUTHORIZED	DM.SECURE_LINK
Unmitigated Impact	Very High	Very High	Very High	Very High
Unmitigated Likelihood	High	High	High	High
Unmitigated Risk	Very High	Very High	Very High	Very High

Mitigating Actions

M.MAC. **Transfer** to the implementation:

- For DM.PROTECTED and DM.EXPOSED, if the implementation applies a MAC, a subsequent read can detect that data had not been written correctly. However, MAC's are not error correcting, therefore the implementation can only mark the data as corrupt and the data is lost.
- For DM.AUTHORIZED and DM.SECURE_LINK, if the implementation relies on the channel to provide the MAC or tag, there is a brief time of check, time of use (TOCTOU) window, where the storage medium has verified the command but has not written the data to physical storage. If a glitch occurs in this window, and then a subsequent read occurs, the storage medium will apply a new tag to a reply containing corrupt data, and the storage service will not be aware that that data returned has been corrupted. However, if the storage service applies a MAC before submitting the command, it can detect, but not correct, this corruption.

M.ErrorCorrectingCoding. **Transfer** to the implementation: for all deployment models, if the storage medium uses error correcting codes (ECC), it can detect and correct a certain number of incorrect bits in the data it reads back — at the expense of extra storage. If the storage medium does not offer ECC capability, the storage service could apply it and verify the coding in software, although this is generally less efficient than hardware.

M.GlitchDetection. **Transfer** to the implementation: for all deployment models, glitch detection can reduce the risk of a successful glitch.

M.ReadAfterWrite. **Transfer** to the implementation: for all deployment models, perform a checked-read after a write in the storage service. The storage service can perform a read operation immediately after a write, while it still retains the original value in memory, and compare the two before reporting a successful write. However, this has performance challenges: therefore, the implementation can decide to do this on a sampling basis.

Residual Impact	Very High	Very High	High	High
Residual Likelihood	Low	Very Low	Very Low	Very Low
Residual Risk	Medium	Low	Low	Low

B.4 Mitigation Summary

This section provides a summary of the mitigations described in the threat analysis, organized by the entity responsible for providing the mitigation.

B.4.1 Architecture level mitigations

Table 10 lists the mitigations that are controlled by the architecture.

Table 10 Mitigations that are **controlled** by the Architecture

Mitigations	Description	Threats
M.MemoryBuffer	In all deployment models, input buffers are fully consumed by the implementation before returning from a function.	T.INTERFACE_ABUSE

B.4.2 Implementation-level mitigations

Table 11 lists the mitigations that are transferred to the implementation. These are also known as 'remediations'.

Table 11 Mitigations that are **transferred** to the implementation

Mitigations	Description	Threats
M.AntiRollback	When using DM.EXPOSED, the implementation must provide a mechanism to prevent an attacker from replacing the stored data with a version that was valid at a previous date. An attacker can use this attack to reinstate flawed firmware, or to return to a version with a broken credential.	T.DIRECT_WRITE
M.AuthenticateEndpoints	When using DM.AUTHORIZED or DM.SECURE_LINK, the storage service must authenticate the storage medium before reading from it or writing to it.	T.MITM
M.Encrypt	When using DM.EXPOSED or DM.AUTHORIZED, the storage service must encrypt data to be written to storage, and decrypt data read from storage, inside the isolated environment to ensure confidentiality.	T.EAVESDROPPING, T.MITM, T.DIRECT_READ, T.DIRECT_WRITE
M.ErrorCorrectingCoding	In all deployments, to deter attacks based on glitching the power or clock, the implementation can implement error correcting coding on stored data.	T.GLITCH_WRITE

continues on next page

Table 11 – continued from previous page

Mitigations	Description	Threats
M.FullyQualifiedNames	In all deployments, the implementation must identify which caller each stored object belongs to and must refer to them internally by the combination of caller identity and name. Otherwise, it might return a stored object to the wrong caller.	T.SPOOF_READ, T.SPOOF_WRITE
M.ImplicitIdentity	In all deployments, the implementation must identify the caller.	T.SPOOF_READ, T.SPOOF_WRITE
M.GlitchDetection	In all deployments, to deter attacks based on glitching the power or clock, the implementation can implement detection circuits.	T.GLITCH_READ, T.GLITCH_WRITE
M.MAC	In DM.EXPOSED, the storage service must apply an integrity check, a MAC, signature, or authenticated encryption tag, within the storage service before it is sent to storage. It must also verify this on every read.	T.MITM, T.DIRECT_WRITE, T.REPLACE
M.PRoTRootedSecLink	In DM.SECURE_LINK, the storage service must use a secure channel rooted within the isolated environment to ensure there is no opportunity for eavesdropping.	T.EAVESDROPPING
M.ReadAfterWrite	To deter glitch attacks on writing data, the implementation can read the data it has just written to verify it.	T.GLITCH_WRITE
M.ReplayProtection	In DM.AUTHORIZED and DM.SECURE_LINK there must be protection against an attacker replaying previous messages.	T.DIRECT_READ, T.DIRECT_WRITE
M.UniqueKeys	In DM.AUTHORIZED and DM.SECURE_LINK the keys used by the storage service and storage medium must be unique, otherwise there is no mechanism for detecting that the storage medium has been replaced.	T.MITM, T.REPLACE
M.ValidateParameter	In all deployment models, check all API parameters to lie within valid ranges, including memory access permissions.	T.INTERFACE_ABUSE
M.VerifyReplies	In DM.AUTHORIZED and DM.SECURE_LINK the storage service must verify all replies from the partition that implements storage, to ensure that they do indeed come from the expected partition and no errors are reported.	T.MITM, T.REPLACE

B.4.3 User-level mitigations

Table 12 lists mitigations that are transferred to the application or other external components. These are also known as 'residual risks'.

Table 12 Mitigations that are **transferred** to the application

Mitigations	Description	Threats
M.UseSecurePartitions	In all deployments, if the caller wants to be certain that there is no chance of eavesdropping, they should make use of caller isolation, with each caller in its own isolated partition.	T.EAVESDROPPING

B.4.4 Mitigations required by each deployment model

Table 13 summarizes the mitigations required in each deployment model.

Table 13 Mitigations required by each deployment model

Implementation	Mitigations
DM.PROTECTED	M.ErrorCorrectingCoding, M.FullyQualifiedNames, M.GlitchDetection, M.ImplicitIdentity, M.MemoryBuffer, M.ReadAfterWrite, M.UseSecurePartitions, M.ValidateParameter
DM.EXPOSED	M.AntiRollback, M.Encrypt, M.ErrorCorrectingCoding, M.FullyQualifiedNames, M.GlitchDetection, M.ImplicitIdentity, M.MAC, M.MemoryBuffer, M.ReadAfterWrite, M.UseSecurePartitions, M.ValidateParameter
DM.AUTHORIZED	M.AuthenticateEndpoints, M.ErrorCorrectingCoding, M.FullyQualifiedNames, M.GlitchDetection, M.ImplicitIdentity, M.MemoryBuffer, M.ReadAfterWrite, M.ReplayProtection, M.UniqueKeys, M.UseSecurePartitions, M.VerifyReplies, M.ValidateParameter
DM.SECURE_LINK	M.AuthenticateEndpoints, M.ErrorCorrectingCoding, M.FullyQualifiedNames, M.GlitchDetection, M.ImplicitIdentity, M.MemoryBuffer, M.PRoTRootedSecLink, M.ReadAfterWrite, M.ReplayProtection, M.UniqueKeys, M.UseSecurePartitions, M.VerifyReplies, M.ValidateParameter

In implementations DM.PROTECTED and DM.SECURE_LINK, the stored data can be implicitly trusted, and therefore it is not required to be encrypted or authenticated. There is no more secure location to store verification data, therefore, any attacker able to access the stored data would also be able to access the key. However, it is possible for the data to be accidentally corrupted, therefore standard engineering practice to guard against this, for example the use of error correcting codes, should be used.

In implementation DM.EXPOSED, the data can be read or modified by an attacker, therefore the storage service must provide confidentiality, integrity, and authenticity by cryptographic means. The keys used to do this must be stored securely. This could be a key derived from the HUK, or separately stored in fuse in a location only readable from the *PRoT*.

As the attacker can always read and modify the stored data, even if they cannot decrypt the data, they can attempt to subvert a change by resetting the storage medium to a prior state. To detect this, the storage service needs to have some means of authenticating that it is reading the most recent state. This implies some form of authentication data stored in a location the attacker cannot modify.

In implementation DM.AUTHORIZED, the data can be observed, even if it cannot be modified. Therefore, data stored does need to be encrypted for confidentiality. However, provided the authentication protocol is strong, and prevents replay, it should not be possible for an attacker to modify the stored data. As the store applies a MAC to each reply, the storage service does not need to apply extra integrity.

In implementation DM.SECURE_LINK provided the secure channel is rooted within the *PRoT*, the data transferred cannot be observed, and any modification will be detected. Therefore, no further encryption is needed for confidentiality or integrity.

Appendix C: 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 v. Documentation clarifications.
2023-03-23	*1.0.2 Rel*	Clarified the protection requirements for ITS. See *Internal Trusted Storage requirements* on page 15. Fixed inconsistent descriptions of `PSA_ERROR_STORAGE_FAILURE` errors.
2024-01-22	*1.0.3 Rel*	Introduced a Security Risk Assessment. See *Security Risk Assessment* on page 35.

Index of API elements

PSA_I

PSA_ITS_API_VERSION_MAJOR, 19
PSA_ITS_API_VERSION_MINOR, 19
psa_its_get, 21
psa_its_get_info, 22
psa_its_remove, 23
psa_its_set, 20

PSA_P

PSA_PS_API_VERSION_MAJOR, 23
PSA_PS_API_VERSION_MINOR, 24
psa_ps_create, 28
psa_ps_get, 25
psa_ps_get_info, 26
psa_ps_get_support, 30
psa_ps_remove, 27
psa_ps_set, 24
psa_ps_set_extended, 29

PSA_S

PSA_STORAGE_FLAG_NONE, 18
PSA_STORAGE_FLAG_NO_CONFIDENTIALITY, 19
PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION, 19
PSA_STORAGE_FLAG_WRITE_ONCE, 18
PSA_STORAGE_SUPPORT_SET_EXTENDED, 19
psa_storage_create_flags_t, 18
psa_storage_info_t, 18
psa_storage_uid_t, 18