PSA Certified Attestation API 1.0

Document number:	IHI 0085
Release Quality:	Final
Issue Number:	4
Confidentiality:	Non-confidential
Date of Issue:	23/09/2025

Contents

About this document

Release information

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

Table 1 Document revision history

Date	Version	Confidentiality	Change
February 2019	1.0 beta 0	Non-confidential	Initial publication.
June 2019	1.0.0	Non-confidential	First stable release with 1.0 API finalized. Uses the PSA Certified API common error status codes. Modified the API parameters to align with other PSA Certified APIs. Updated the claims and lifecycle to match the latest Platform Security Model. Updated CBOR example in the appendix.
August 2019	1.0.1	Non-confidential	Recommend type byte 0x01 for arm_psa_UEID. Remove erroneous guidance regarding EAT's origination claim.
February 2020	1.0.2	Non-confidential	Clarify the claim number of Instance ID. Permit COSE-Mac0 for signing tokens (with appropriate warning). Update URLs.
October 2022	1.0.3	Non-confidential	Relicensed as open source under CC BY-SA 4.0. CDDL definition added to the appendices. Example header file added to the appendices. Minor corrections and clarifications.
September 2025	1.0.4	Non-confidential	GlobalPlatform governance of PSA Certified evaluation scheme.

The detailed changes in each release are described in *Document history* on page 30.

PSA Certified Attestation API

License

Text and illustrations

About the license

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

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

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

Source code

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[PSM]	ARM DEN 0128	*Platform Security Model*. developer.arm.com/documentation/den0128
[PSA-STAT]	ARM IHI 0097	*PSA Certified Status code API*. arm-software.github.io/psa-api/status-code
[PSA-FF-M]	ARM DEN 0063	*Arm® Platform Security Architecture Firmware Framework*. pages.arm.com/psa-apis
[C99]		ISO/IEC, *ISO/IEC 9899:1999 — Programming Languages — C*, December 1999. www.iso.org/standard/29237.html
[EAT]		*IETF Entity Attestation Token (EAT)*, Draft. datatracker.ietf.org/doc/draft-ietf-rats-eat
[PSATOKEN]		*Arm's Platform Security Architecture (PSA) Attestation Token*, Draft. datatracker.ietf.org/doc/draft-tschofenig-rats-psa-token
[STD94]		Bormann, C. and P. Hoffman, *Concise Binary Object Representation (CBOR)*, December 2020. rfc-editor.org/info/std94
[STD96]		Schaad, J., *CBOR Object Signing and Encryption (COSE): Structures and Process*, August 2022. rfc-editor.org/info/std96
[RFC8610]		IETF, *Concise Data Definition Language (CDDL)*. tools.ietf.org/html/rfc8610
[EAN-13]		*International Article Number*. www.gs1.org/standards/barcodes/ean-upc

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 3 Terms and abbreviations

Term	Meaning
CBOR	See *Concise Binary Object Representation*.
Concise Binary Object Representation (CBOR)	A format for encoding binary objects in a bitstream, defined in *Concise Binary Object Representation (CBOR)* [STD94].
EAT	See *Entity Attestation Token*.

continues on next page

Table 3 – continued from previous page

Term	Meaning
Entity Attestation Token (EAT)	A report format for attestation tokens, defined in *IETF Entity Attestation Token (EAT)* [EAT].
IAK	See *Initial Attestation Key*.
Immutable Platform Root of Trust	Part of the *Platform Root of Trust*, which is inherently trusted. This refers to the hardware and firmware that cannot be updated on a production device. See *Platform Security Model* [PSM].
IMPLEMENTATION DEFINED	Behavior that is not defined by this specification, but is defined and documented by individual implementations. Application developers can choose to depend on IMPLEMENTATION DEFINED behavior, but must be aware that their code might not be portable to another implementation.
Initial Attestation Key (IAK)	Typically, the Initial Attestation Key is a secret private key from an asymmetric key-pair accessible only to the Initial Attestation service within the *Platform Root of Trust*. See *Platform Security Model* [PSM].
Non-secure Processing Environment (NSPE)	This is the security domain outside of the *Secure Processing Environment*. It is the application domain, typically containing the application firmware and hardware.
NSPE	See *Non-secure Processing Environment*.
Platform Root of Trust (PRoT)	The overall trust anchor for the system. This ensures the platform is securely booted and configured, and establishes the secure environments required to protect security services. See *Platform Security Model* [PSM].
PRoT	See *Platform Root of Trust*.
PSA	Platform Security Architecture
Secure Processing Environment (SPE)	This is the security domain that includes the *Platform Root of Trust* domain.
SPE	See *Secure Processing Environment*.

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.

Current status and anticipated changes

The token format defined within this specification has been superseded by the attestation token format defined in *Arm's Platform Security Architecture (PSA) Attestation Token* [PSATOKEN]. A future update to this specification will incorporate the new token definition.

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 (Attestation API).
- The number and issue (IHI 0085 1.0.4).
- 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 GlobalPlatform's PSA Certified evaluation scheme 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 and www.trustedfirmware.org.

1.2 About the Attestation API

The interface described in this document is a PSA Certified API, that provides a verifiable report of the state of the platform. The platform attestation service is provided by the *Platform Root of Trust* and is described in *Platform Security Model* [PSM].

This document includes:

- A set of common use cases. See *Use cases and rationale* on page 10.
- Information about the attestation report and the format. See *Initial Attestation report* on page 12.
- The associated Application Programming Interface (API). See *API reference* on page 18.

The Attestation API can be used either to directly produce verifiable evidence about the platform state in the context of a challenge-response interaction, or as a way to bootstrap trust in other attestation schemes. The PSA Certified framework provides the generic security features allowing OEM and service providers to integrate various attestation schemes on top of the Platform Root of Trust.

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

2 Use cases and rationale

The following subsections describe the primary use cases that this version of Attestation API aims to support. Other use cases are also possible.

The *Platform Root of Trust* (PRoT) reports information, known as claims, that can be used to determine the exact implementation of the PRoT and its security state. If the PRoT loads other components then it also includes information about what it has loaded. Other components outside of the PRoT can add additional information to the report by calling the provided API, which will include and sign the additional information. The PRoT signs attestation reports using the *Initial Attestation Key* (IAK).

2.1 Device enrolment

Enrolment is the ability for an online service to enlist a device. For example, a generic connected sensor that becomes part of a company's deployment. As part of the enrolment process, credentials need to be created for each device. However, the devices themselves need to be trustworthy to ensure that credentials are not leaked.

A common solution to this problem is to certify security hardware using third-party labs, who are trusted to deliver worthwhile certifications. By placing trust in evaluation reports (such as Common Criteria or PSA Certified), one can ascertain whether a Root of Trust exhibits important security properties. For example, one important property is the ability to generate a key pair of good quality (using a non-predictable random number generator) and store it in an isolated and tamper-proof area, which provides strong assurance that a device private key is only ever known by that device. Each device instance contains a protected attestation key that can be used to prove that they are a particular certified implementation.

During such an enrolment process, a device might generate a new key pair and create a Certificate Signing Request (CSR) or equivalent, containing:

- The public key of the generated key-pair.
- A proof of possession of the corresponding private key (in general this is the public key signed by the private key). This protects against man-in-the-middle attacks where an attacker can hijack the enrolment to insert their own public key into the device request.
- An initial attestation, in order for the recipient to assess how that particular combination of hardware and firmware can be trusted.

The CSR is then passed to a Certification Authority who can assign it an identity with the new service and then return an identity certificate signed using the private key of the Certification Authority. The Certification Authority may be operated by the company who owns the devices or operated by a trusted third party. Creating extra identities on devices is expected to be a routine operation.

If a device enforces a high level of isolation, where all applications execute within their own Secure Partition, then it allows several mutually-distrustful providers to install their applications side-by-side without having to worry about leaking assets from one application to another.

The attestation identity can be verified in an attestation process and checked against certification information. At the end of the process the credential manager can establish a secure connection to the attested endpoint, and deliver credentials. For example, these may be service access credentials.

2.2 Identifying certification

The combination of a hardware entity and the software installed on that entity can be certified to conform to some published security level.

Manufacturers of devices can advertise a security certification as an incentive to purchase their devices, or because it is a requirement from a regulator. To gain the certification a manufacturer can engage a test lab to verify the hardware and software combination of a device conforms to specific standards. Certification should not be declared by the device, instead it is a dynamic situation where the hardware and software state can be checked against the current known certification status for that combination.

The initial attestation report declares the state of the device to a verification service. The verification service can then:

- Verify the production status of the device identity. For example, to identify whether the device is in an inventory, and whether it is a secured production device or a development device.
- Verify the certification status of a device. This involves checking that all components are up to date, correctly signed, and certified to work together.

2.3 Integrity reporting

A party may want to check the received list of claims against a database of known measurements for each component in order to decide which level of trust should be applied. Additional information can be included, such as the version numbers for all software running on the device. As a minimum, the device provides a hash for each loaded component. Boot measurements are included in order to assess if there are obvious signs of tampering with the device firmware.

Initial attestation requires three services:

- Enrolment verification service enforcing policy as part of service enrolment of the device.
- Production verification service (OEM), providing the production state of an attestation identity
- Certification verification service (third party), verifying that all attested components are up to date, signed correctly, and certified to work together.

It is possible to further separate these roles. For example, there may be a separate software verification service.

These services can be hosted by different parties in online or offline settings:

- The first service requires generating a challenge, reading back the device's token, and validating the signature of the token.
- The second service may periodically log the current security state for all addressable devices and make that information available upon request. It does not require the knowledge of any pre-shared secret or a prior trust exchange with a device vendor. The various databases required for the full verification process may be local, replicated, or centralized, depending on the particular market.

Further information about using existing attestation protocols can be found in [PSM].

3 Initial Attestation report

This section begins with a description of the information model for the report and then describes the expected format.

3.1 Information model

The following table describes the mandatory and optional claims in the report:

Claim	Mandatory	Description
Auth challenge	Yes	Input object from the caller. For example, this can be a cryptographic nonce or a hash of locally attested data. The length must be 32, 48, or 64 bytes. This is the `auth_challenge` parameter to `psa_initial_attest_get_token()`.
Instance ID	Yes	Represents the unique identifier of the instance: • When using an asymmetric key-pair for the *Initial Attestation Key* (IAK), Arm recommends the Instance ID be a hash of the corresponding public key. • When using a symmetric key for the IAK, Arm recommends that the Instance ID is always a double hash of the key, hence `InstanceID = H(H(IAK))`. This eliminates risks when exposing the key to different HMAC block size. For further information, read RFC2104. The use of the IAK is also discussed in [PSM].
Verification service indicator	No	A hint used by a relying party to locate a validation service for the token. The value is a text string that can be used to locate the service or a URL specifying the address of the service. A verifier may choose to ignore this claim in favor of other information.
Profile definition	No	Contains the name of a document that describes the 'profile' of the report. The document name may include versioning. The value for this specification is **PSA_IOT_PROFILE_1**.
Implementation ID	Yes	Uniquely identifies the underlying *Immutable Platform Root of Trust*. A verification service can use this claim to locate the details of the verification process. Such details include the implementation's origin and associated certification state. The full definition is in [PSM].
Client ID	Yes	Represents the Partition ID of the caller. It is a signed integer whereby negative values represent callers from the *NSPE* and where positive IDs represent callers from the *SPE*. The value `0` is not permitted. The full definition of a Partition ID is provided by *Arm® Platform Security Architecture Firmware Framework* [PSA-FF-M]. It is essential that this claim is checked in the verification process to ensure that a security domain cannot spoof a report from another security domain.
Security Lifecycle	Yes	Represents the current lifecycle state of the *Platform Root of Trust* (PRoT). The state is represented by an integer that is partitioned to convey a major state and a minor state. The major state is mandatory and defined by [PSM]. The minor state is optional and IMPLEMENTATION DEFINED. The PRoT security lifecycle state and implementation state are encoded as follows: • version[15:8] — PRoT security lifecycle state • version[7:0] — IMPLEMENTATION DEFINED state.

3.1.1 Software components

Each software component in the Software Components claim must include the required properties of the following table:

Key ID	Type	Required	Description
1	Measurement type	No	A short string representing the role of this software component (e.g. 'BL' for boot loader). Expected types may include: • BL (a bootloader) • PRoT (a component of the Platform Root of Trust) • ARoT (a component of the Application Root of Trust) • App (a component of the NSPE application) • TS (a component of a trusted subsystem)
2	Measurement value	Yes	Represents a hash of the invariant software component in memory at startup time. The value must be a cryptographic hash of 256 bits or stronger.
3	Reserved	No	Reserved
4	Version	No	The issued software version in the form of a text string. The value of this claim corresponds to the entry in the original signed manifest of the component. This field must be present to be compliant with [PSM].
5	Signer ID	No	The hash of a signing authority public key for the software component. The value of this claim corresponds to the entry in the original manifest for the component. This can be used by a verifier to ensure the components were signed by an expected trusted source. This field must be present to be compliant with [PSM].

continues on next page

Table 4 – continued from previous page

Key ID	Type	Required	Description
6	Measurement description	No	Description of the software component, which represents the way in which the measurement value of the software component is computed. The value is a text string containing an abbreviated description (or name) of the measurement method which can be used to lookup the details of the method in a profile document. This claim may normally be excluded, unless there is an exception to the default measurement described in the profile for a specific component.

3.2 Report format and signing

This section describes the specific representation, encoding and signing of the information described in the Information Model.

3.2.1 Token encoding

The report is represented as a token, which must be formatted in accordance to *IETF Entity Attestation Token (EAT)* [EAT] draft specification. The token consists of a series of claims declaring evidence as to the nature of the instance of hardware and software. The claims are encoded with the *CBOR* format, defined in *Concise Binary Object Representation (CBOR)* [STD94].

3.2.2 Signing

The token is signed following the structure defined in *CBOR Object Signing and Encryption (COSE): Structures and Process* [STD96] specification:

- For asymmetric key algorithms, the signature structure must be COSE-Sign1. An asymmetric key algorithm is needed to achieve all the use cases defined in *Use cases and rationale* on page 10.
- For symmetric key algorithms, the structure must be COSE-Mac0.

> **Warning**
>
> A symmetric key is **strongly discouraged** due to the associated infrastructure costs for key management and operational complexities. It may also restrict the ability to interoperate with scenarios that involve third parties (see *Use cases and rationale* on page 10).

3.2.3 EAT standard claims

The token is modelled to include custom values that correspond to the following EAT standard claims (as expressed in the draft EAT proposal):

- **nonce** (mandatory); arm_psa_nonce is used instead
- **UEID** (mandatory); arm_psa_UEID is used instead

A future version of the profile, corresponding to an issued standard, might declare support for both custom and standard claims as a transitionary state towards exclusive use of standard claims.

3.2.4 EAT custom claims

The token can include the following EAT custom claims. Custom claims for the Attestation API have a root identity of -75000.

Some fields must be at least 32 bytes to provide sufficient cryptographic strength.

Key ID	Type	Name	CBOR type
-75000	Profile Definition	`arm_psa_profile_id`	Text string
-75001	Client ID	`arm_psa_partition_id`	Unsigned integer or Negative integer
-75002	Security Lifecycle	`arm_psa_security_lifecycle`	Unsigned integer
-75003	Implementation ID	`arm_psa_implementation_id`	Byte string (>=32 bytes)
-75004	Boot seed	`arm_psa_boot_seed`	Byte string (>=32 bytes)
-75005	Hardware version	`arm_psa_hw_version`	Text string
-75006	Software components (compound map claim)	`arm_psa_sw_components`	Array of map entries. The map entries have the following types: 1. Text string (type) 2. Byte string (measurement, >=32 bytes) 3. Reserved 4. Text string (version) 5. Byte string (signer ID, >=32 bytes) 6. Text string (measurement description) See *Software components* on page 14 for details.
-75007	No software measurements	`arm_psa_no_sw_measurements`	Unsigned integer (the recommended value is 1)
-75008	Auth challenge	`arm_psa_nonce`	Byte string

continues on next page

Table 5 – continued from previous page

Key ID	Type	Name	CBOR type
-75009	Instance ID	`arm_psa_UEID`	Byte string (the type byte should be set to `0x01`. The type byte is described in the [EAT] draft.)
-75010	Verification service indicator	`arm_psa_origination`	Text string

An example report can be found in *Example report* on page 24.

4 API reference

The Attestation API defines a header file that is provided by the implementation. The header is `psa/initial_attestation.h`.

All the elements are defined in the C language. The Attestation API makes use of standard C data types, including the fixed-width integer types from the ISO C99 specification update [C99].

4.1 API conventions

All functions return a status indication of type `psa_status_t`, which is defined by *PSA Certified Status code API* [PSA-STAT]. The value `0` (`PSA_SUCCESS`) indicates successful operation, and a negative value indicates an error. Each API documents the specific error codes that might be returned, and the meaning of each error.

All parameters of pointer type must be valid, non-null pointers unless the pointer is to a buffer of length 0 or the function's documentation explicitly describes the behavior when the pointer is null. For implementations where a null pointer dereference usually aborts the application, passing NULL as a function parameter where a null pointer is not allowed should abort the caller in the habitual manner.

Pointers to input parameters may be in read-only memory. Output parameters must be in writable memory. Output parameters that are not buffers must also be readable, and the implementation must be able to write to a non-buffer output parameter and read back the same value.

4.2 Status codes

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

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

```
typedef int32_t psa_status_t;

#define PSA_SUCCESS ((psa_status_t)0)

#define PSA_ERROR_GENERIC_ERROR         ((psa_status_t)-132)
#define PSA_ERROR_INVALID_ARGUMENT      ((psa_status_t)-135)
#define PSA_ERROR_BUFFER_TOO_SMALL      ((psa_status_t)-138)
#define PSA_ERROR_SERVICE_FAILURE       ((psa_status_t)-144)
```

These definitions must be available to an application that includes the `psa/initial_attestation.h` header file.

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

4.3 General definitions

4.3.1 PSA_INITIAL_ATTEST_API_VERSION_MAJOR (macro)

The major version of this implementation of the Attestation API.

```
#define PSA_INITIAL_ATTEST_API_VERSION_MAJOR 1
```

4.3.2 PSA_INITIAL_ATTEST_API_VERSION_MINOR (macro)

The minor version of this implementation of the Attestation API.

```
#define PSA_INITIAL_ATTEST_API_VERSION_MINOR 0
```

4.3.3 PSA_INITIAL_ATTEST_MAX_TOKEN_SIZE (macro)

The maximum possible size of a token.

```
#define PSA_INITIAL_ATTEST_MAX_TOKEN_SIZE /* implementation-specific value */
```

The value of this constant is IMPLEMENTATION DEFINED.

4.4 Challenge sizes

The following constants define the valid challenge sizes that must be supported by the function `psa_initial_attest_get_token()` and `psa_initial_attest_get_token_size()`.

An implementation must not support other challenge sizes.

4.4.1 PSA_INITIAL_ATTEST_CHALLENGE_SIZE_32 (macro)

A challenge size of 32 bytes (256 bits).

```
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_32 (32u)
```

4.4.2 PSA_INITIAL_ATTEST_CHALLENGE_SIZE_48 (macro)

A challenge size of 48 bytes (384 bits).

```
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_48 (48u)
```

4.4.3 PSA_INITIAL_ATTEST_CHALLENGE_SIZE_64 (macro)

A challenge size of 64 bytes (512 bits).

```
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_64 (64u)
```

4.5 Attestation

4.5.1 `psa_initial_attest_get_token` (function)

Retrieve the Initial Attestation Token.

```
psa_status_t psa_initial_attest_get_token(const uint8_t *auth_challenge,
                                          size_t challenge_size,
                                          uint8_t *token_buf,
                                          size_t token_buf_size,
                                          size_t *token_size);
```

Parameters

`auth_challenge`	Buffer with a challenge object. The challenge object is data provided by the caller. For example, it may be a cryptographic nonce or a hash of data (such as an external object record). If a hash of data is provided then it is the caller's responsibility to ensure that the data is protected against replay attacks (for example, by including a cryptographic nonce within the data).
`challenge_size`	Size of the buffer `auth_challenge` in bytes. The size must always be a supported challenge size. Supported challenge sizes are defined in *Challenge sizes* on page 19.
`token_buf`	Output buffer where the attestation token is to be written.
`token_buf_size`	Size of `token_buf`. The expected size can be determined by using the `psa_initial_attest_get_token_size` function.
`token_size`	Output variable for the actual token size.

Outputs

`*token_buf`	On success, the attestation token.
`*token_size`	On success, the number of bytes written into `token_buf`.

Returns: `psa_status_t`

`PSA_SUCCESS`	Action was performed successfully.
`PSA_ERROR_SERVICE_FAILURE`	The implementation failed to fully initialize.
`PSA_ERROR_BUFFER_TOO_SMALL`	`token_buf` is too small for the attestation token.
`PSA_ERROR_INVALID_ARGUMENT`	The challenge size is not supported.
`PSA_ERROR_GENERIC_ERROR`	An unspecified internal error has occurred.

Description

Retrieves the Initial Attestation Token. A challenge can be passed as an input to mitigate replay attacks.

4.5.2 `psa_initial_attest_get_token_size` (function)

Calculate the size of an Initial Attestation Token.

```
psa_status_t psa_initial_attest_get_token_size(size_t challenge_size,
                                               size_t *token_size);
```

Parameters

`challenge_size`	Size of a challenge object in bytes. This must be a supported challenge size as specified in *Challenge sizes* on page 19.
`token_size`	Output variable for the token size.

Outputs

`*token_size`	On success, the maximum size of an attestation token in bytes when using the specified `challenge_size`

Returns: `psa_status_t`

`PSA_SUCCESS`	Action was performed successfully.
`PSA_ERROR_SERVICE_FAILURE`	The implementation failed to fully initialize.
`PSA_ERROR_INVALID_ARGUMENT`	The challenge size is not supported.
`PSA_ERROR_GENERIC_ERROR`	An unspecified internal error has occurred.

Description

Retrieve the exact size of the Initial Attestation Token in bytes, given a specific challenge size.

Appendix A: Example header file

Each implementation of the Attestation API must provide a header file named `psa/initial_attestation.h`, in which the interface elements in this specification are defined.

This appendix provides a example of the `psa/initial_attestation.h` header file with all of the API elements. This can be used as a starting point or reference for an implementation.

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

A.1 psa/inital_attestation.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Attestation API v1.0
 */

#ifndef PSA_INITIAL_ATTESTATION_H
#define PSA_INITIAL_ATTESTATION_H

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

#ifdef __cplusplus
extern "C" {
#endif

#define PSA_INITIAL_ATTEST_API_VERSION_MAJOR 1
#define PSA_INITIAL_ATTEST_API_VERSION_MINOR 0
#define PSA_INITIAL_ATTEST_MAX_TOKEN_SIZE /* implementation-specific value */
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_32 (32u)
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_48 (48u)
#define PSA_INITIAL_ATTEST_CHALLENGE_SIZE_64 (64u)
psa_status_t psa_initial_attest_get_token(const uint8_t *auth_challenge,
                                          size_t challenge_size,
                                          uint8_t *token_buf,
                                          size_t token_buf_size,
                                          size_t *token_size);
psa_status_t psa_initial_attest_get_token_size(size_t challenge_size,
                                               size_t *token_size);
```

(continues on next page)

(continued from previous page)

```
#ifdef __cplusplus
}
#endif

#endif // PSA_INITIAL_ATTESTATION_H
```

Appendix B: Example report

An example report is included here in extended *CBOR* diagnostic form for illustrative purposes:

```
18(
[
/ protected / h'a10126' / {
    \ alg \ 1: -7 \ ECDSA 256 \
  } / ,
/ unprotected / {},
/ payload / h'a93a000124fb582000010203040506070809 0a0b0c0d0e0f1011121
31415161718191a1b1c1d1e1f3a000124fa582000010203040506070809 0a0b0c0d0e
0f101112131415161718191a1b1c1d1e1f3a000124fd84a402582000010203040506 0
708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0465332e312e34055820
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f01624
24ca402582000010203040506070809 0a0b0c0d0e0f101112131415161718191a1b1c
1d1e1f0463312e3105582000010203040506070809 0a0b0c0d0e0f101112131415161
718191a1b1c1d1e1f016450526f54a402582000010203040506070809 0a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f0463312e3005582000010203040506070809
a0b0c0d0e0f101112131415161718191a1b1c1d1e1f016441526f54a4025820000102
030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0463322e320
558200001020304050607080 90a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
01634170703a000124f91930003a000124ff582000010203040506070809 0a0b0c0d0
e0f101112131415161718191a1b1c1d1e1f3a000125016c7073615f76657269666965
723a000124f8203a000125005821010001020304050607080 90a0b0c0d0e0f1011121
31415161718191a1b1c1d1e1f3a000124f7715053415f496f545f50524f46494c455f
31' / {
   / arm_psa_boot_seed / -75004: h'000102030405060708090a0b0c0d0e0f10
   111213141516171 8191a1b1c1d1e1f',
   / arm_psa_implementation_id / -75003: h'000102030405060708090a0b0c
   0d0e0f101112131415161718191a1b1c1d1e1f',
   / arm_psa_sw_components / -75006: [
        {
          / measurement / 2: h'000102030405060708090a0b0c0d0e0f101112
          131415161718191a1b1c1d1e1f',
          / version / 4: "3.1.4",
          / signerID / 5: h'000102030405060708090a0b0c0d0e0f101112131
          415161718191a1b1c1d1e1f',
          / type / 1: "BL"
        },
        {
          / measurement / 2: h'000102030405060708090a0b0c0d0e0f101112
          131415161718191a1b1c1d1e1f',
          / version / 4: "1.1",
```

(continues on next page)

(continued from previous page)

```
          / signerID / 5: h'000102030405060708090a0b0c0d0e0f101112131
          415161718191a1b1c1d1e1f',
          / type / 1: "PRoT"
        },
        {
          / measurement / 2: h'000102030405060708090a0b0c0d0e0f101112
          1314151617181 91a1b1c1d1e1f',
          / version / 4: "1.0",
          / signerID / 5: h'000102030405060708090a0b0c0d0e0f101112131
          415161718191a1b1c1d1e1f',
          / type / 1: "ARoT"
        },
        {
          / measurement / 2: h'000102030405060708090a0b0c0d0e0f101112
          1314151617181 91a1b1c1d1e1f',
          / version / 4: "2.2",
          / signerID / 5: h'000102030405060708090a0b0c0d0e0f101112131
          415161718191a1b1c1d1e1f',
          / type / 1: "App"
        }
      ],
    / arm_psa_security_lifecycle / -75002: 12288 / SECURED /,
    / arm_psa_nonce / -75008: h'000102030405060708090a0b0c0d0e0f10111
    21314151617181 91a1b1c1d1e1f',
    / arm_psa_origination / -75010: "psa_verifier",
    / arm_psa_partition_id / -75001: -1,
    / arm_psa_UEID / -75009: h'010001020304050607080 90a0b0c0d0e0f1011
    121314151617181 91a1b1c1d1e1f',
    / arm_psa_profile_id / -75000: "PSA_IOT_PROFILE_1"
  }),
  } / ,
/ signature / h'58860508ee7e8cc48eba872dbb5d694a542b1322ad0d51023c197
0df429f06501c683a95108a0cced0a6e80e0966f22bd63d1c0056974a11ba332d7877
87fb4f'
]
)
```

Appendix C: CDDL

The *Concise Data Definition Language (CDDL)* [RFC8610] definition of the PSA token is included here for reference:

```
psa-token = {
    psa-nonce-claim,
    psa-client-id,
    psa-instance-id,
    psa-implementation-id,
    psa-hardware-version,
    psa-lifecycle,
    psa-boot-seed,
    ( psa-software-components // psa-no-sw-measurement ),
    psa-profile,
    psa-verification-service-indicator,
}

arm_psa_profile_id = -75000
arm_psa_partition_id = -75001
arm_psa_security_lifecycle = -75002
arm_psa_implementation_id = -75003
arm_psa_boot_seed = -75004
arm_psa_hw_version = -75005
arm_psa_sw_components = -75006
arm_psa_no_sw_measurements = -75007
arm_psa_nonce = -75008
arm_psa_UEID = -75009
arm_psa_origination = -75010

psa-boot-seed-type = bytes .size 32

psa-hash-type = bytes .size 32 / bytes .size 48 / bytes .size 64


psa-boot-seed = (
    arm_psa_boot_seed => psa-boot-seed-type
)

psa-client-id-nspe-type = -2147483648...0
psa-client-id-spe-type = 1..2147483647

psa-client-id-type = psa-client-id-nspe-type / psa-client-id-spe-type
```

(continues on next page)

(continued from previous page)

```
psa-client-id = (
    arm_psa_partition_id => psa-client-id-type
)

psa-hardware-version-type = text .regexp "[0-9]{13}"

psa-hardware-version = (
    ? arm_psa_hw_version => psa-hardware-version-type
)

psa-implementation-id-type = bytes .size 32

psa-implementation-id = (
    arm_psa_implementation_id => psa-implementation-id-type
)

psa-instance-id-type = bytes .size 33

psa-instance-id = (
    arm_psa_UEID => psa-instance-id-type
)

psa-no-sw-measurements-type = 1

psa-no-sw-measurement = (
    arm_psa_no_sw_measurements => psa-no-sw-measurements-type
)

psa-nonce-claim = (
    arm_psa_nonce => psa-hash-type
)

psa-profile-type = "PSA_IOT_PROFILE_1"

psa-profile = (
    ? arm_psa_profile_id => psa-profile-type
)

psa-lifecycle-unknown-type = 0x0000..0x00ff
psa-lifecycle-assembly-and-test-type = 0x1000..0x10ff
psa-lifecycle-psa-rot-provisioning-type = 0x2000..0x20ff
psa-lifecycle-secured-type = 0x3000..0x30ff
psa-lifecycle-non-psa-rot-debug-type = 0x4000..0x40ff
psa-lifecycle-recoverable-psa-rot-debug-type = 0x5000..0x50ff
psa-lifecycle-decommissioned-type = 0x6000..0x60ff

psa-lifecycle-type =
    psa-lifecycle-unknown-type /
```

(continues on next page)

(continued from previous page)

```
    psa-lifecycle-assembly-and-test-type /
    psa-lifecycle-psa-rot-provisioning-type /
    psa-lifecycle-secured-type /
    psa-lifecycle-non-psa-rot-debug-type /
    psa-lifecycle-recoverable-psa-rot-debug-type /
    psa-lifecycle-decommissioned-type

psa-lifecycle = (
    arm_psa_security_lifecycle => psa-lifecycle-type
)

psa-software-component = {
    ? 1 => text,         ; measurement type
    2 => psa-hash-type, ; measurement value
    ? 4  => text,        ; version
    5 => psa-hash-type, ; signer id
    ? 6 => text,         ; measurement description
}

psa-software-components = (
    arm_psa_sw_components => [ + psa-software-component ]
)

psa-verification-service-indicator-type = text

psa-verification-service-indicator = (
    ? arm_psa_origination => psa-verification-service-indicator-type
)
```

Appendix D: Document history

Date	Changes
2019-02-25	*1.0 Beta 0* • First public version for review
2019-06-12	*1.0.0* • First stable release • The API functions now use the shared `psa_status_t` return type. • Error values now use shared 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`. • Some parameter names have been changed to improve legibility. • The description of the Implementation ID claim has been rewritten to better match the definition in PSM. • Signer ID is no longer a mandatory part of the Software Components claim. However, it is needed for PSM compliance. • Explicitly describe which optional claims are required for PSM compliance. • Added lifecycle state (`PSA_LIFECYCLE_ASSEMBLY_AND_TEST`). • Clarifications and improvements to the description of some API elements and to the structure of the document. • Updated CBOR example in the appendix. • Added macro `PSA_INITIAL_ATTEST_MAX_TOKEN_SIZE`.
2019-08-16	*1.0.1* • Fixed typos and descriptions based on feedback. • Recommend type byte 0x01 for arm_psa_UEID. • Remove erroneous guidance regarding EAT's origination claim - it should not be used to find a verification service.
2020-02-06	*1.0.2* • Clarify the claim number of Instance ID • Permit COSE-Mac0 for signing tokens (with appropriate warning) • Update URLs
2022-10-17	*1.0.3* • Relicensed the document under Attribution-ShareAlike 4.0 International with a patent license derived from Apache License 2.0. See *License* on page iv. • Fix CBOR type of `arm_psa_origination` to text string. Spec and example were in conflict, and the example was correct.