Authenticated Debug Access Control 1.0

Architecture & Technology Group

Document number:	DEN 0101
Release Quality:	Final
Issue Number:	1
Confidentiality:	Non-confidential
Date of Issue:	18/05/2022

Contents

About this document

Release information

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

Date	Version	Confidentiality	Change
2022 May	1.0.1 final	Non-confidential	Final release with errata.
2022 Mar	1.0 final	Non-confidential	Final release.
2021 Oct	1.0 beta2	Non-confidential	Draft proposal.
2021 Jun	1.0 beta1	Non-confidential	Draft proposal.
2020 Oct	1.0 beta0	Non-confidential	Draft proposal.
2020 Apr	0.1 Dev0	Confidential	Draft proposal.

For a detailed list of changes, see *Changes in this document* on page 70.

Authenticated Debug Access Control

References

This document refers to the following documents.

Ref	Document Number	Title
[RFC8032]		IETF, *Edwards-Curve Digital Signature Algorithm (EdDSA)*. https://tools.ietf.org/html/rfc8032
[RFC8017]		IETF, *'PKCS #1: RSA Cryptography Specifications Version 2.2'*. https://tools.ietf.org/html/rfc8017
[RFC6979]		IETF, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, 08/2013. https://tools.ietf.org/html/rfc6979
[RFC4493]		IETF, *'The AES-CMAC Algorithm'*. https://tools.ietf.org/html/rfc4493
[RFC2104]	10.17487 / RFC2104	IETF, *'HMAC: Keyed-Hashing for Message Authentication'*. https://tools.ietf.org/html/rfc2104
[RFC6234]		IETF, *'US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)'*. https://tools.ietf.org/html/rfc6234
[X9.62-2005]	ANSI X9.62-2005	American National Standards Institute', *'Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)'*, 11/2005. https://standards.globalspec.com/std/1955141/ANSI%20X9.62
[SECGv2]		Standards for Efficient Cryptography Group, *Recommended Elliptic Curve Domain Parameters*. https://www.secg.org/sec2-v2.pdf
[IHI0076A]	ARM IHI 0076A	Arm Ltd, *Advanced Communications Channel Architecture Specification*, 05/2018. https://developer.arm.com/documentation/ihi0076/a
[SDC600]	101130 / 0002	Arm Ltd, *Arm® CoreSight™ SDC-600 Secure Debug Channel Technical Reference Manual*, 05/2018. https://developer.arm.com/documentation/101130/0002/
[PSA-SM]	ARM DEN 0079	Arm Ltd, *Arm® Platform Security Architecture Security Model*, 02/2019. https://developer.arm.com/architectures/security-architectures/platform-security-architecture/documentation
[ISO-SM2]	ISO ISO/IEC 14888-3:2018	International Organization for Standardization, *IT Security techniques – Digital signatures with appendix – Part 3: Discrete logarithm based mechanisms*, 11/2018. https://www.iso.org/standard/76382.html

Table 1 (continued)

Ref	Document Number	Title
[ISO-SM3]	ISO ISO/IEC 10118-3:2018	International Organization for Standardization, *IT Security techniques – Hash-functions – Part 3: Dedicated hash-functions*, 10/2018. https://www.iso.org/standard/67116.html
[FIPS-186-4]	10.6028 / NIST.FIPS.186-4	National Institute of Standards and Technology, *Federal Information Processing Standards Publication (FIPS PUB) 186-4 – Digital Signature Standard (DSS)*, 07/2013. https://doi.org/10.6028/NIST.FIPS.186-4
[SP800-90A]	10.6028 / NIST.SP.800-90Ar1	'National Institute of Standards and Technology', *NIST Special Publication 800-90A – Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*, 01/2012. https://doi.org/10.6028/NIST.SP.800-90Ar1
[SP800-38B]	10.6028 / NIST.SP.800-38B	National Institute of Standards and Technology, *NIST Special Publication 800-38B- Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*, 05/2005. https://doi.org/10.6028/NIST.SP.800-38B
[SP800-107r1]	10.6028 / NIST.SP.800-107r1	National Institute of Standards and Technology', *NIST Special Publication 800-107r1 – Recommendation for Applications Using Approved Hash Algorithms*, 08/2012. https://doi.org/10.6028/NIST.SP.800-107r1
[Ed25519]		Bernstein et al., *Twisted Edwards curves*, Africacrypt, 2008. https://eprint.iacr.org/2008/013.pdf
[Ed448]		Hamburg, *Ed448-Goldilocks, a new elliptic curve*, NIST ECC Workshop, 2015. https://eprint.iacr.org/2015/625.pdf

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
Access Port (AP)	Access Port
ADAC	See *Authenticated Debug Access Control*.
Advanced Peripheral Bus (APB)	Low speed, low complexity bus for peripherals.
AP	See *Access Port*.
APB	See *Advanced Peripheral Bus*.
API	See *Application Programming Interface*.

Table 2 (continued)

Term	Meaning
Application Programming Interface (API)	A programmable software interface.
Authenticated Debug Access Control (ADAC)	The debug authentication protocol described in this document.
CMSIS	Cortex Microcontroller Software Interface Standard – a vendor independent hardware abstraction layer for Cortex-M processors.
Completer	The Completer is the party responding to the Requester in the protocol. For ADAC this is the device side.
DAP	See *Debug Access Port*.
DCU	See *Debug Control Unit*.
Debug Access Port (DAP)	A block that acts as a Requester on a system bus and provides access to the bus from an external debugger.
Debug Client	Software on the debug host that controls the debug link.
Debug Control Unit (DCU)	Debug Control Unit
Debug Host	The Requester component that performs debug operations on the debug target.
Debug Link	The connection between debug host and debug target through over which the debug client performs debug operations.
Debug Port (DP)	Debug Port
Debug Target	The device-side component which is controlled by the debug host.
Debugger Mailbox	Generic term for a communications channel between a debug host and a software agent running on the device being debugged. The actual hardware IP consists of an AP on the debugger (external) side connected to an APB peripheral on the internal side.
DP	See *Debug Port*.
IC Vendor (ICV)	Silicon Partner (SiP).
ICV	See *IC Vendor*.
IFR	See *Indexed Flash Region*.
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.

Table 2 (continued)

Term	Meaning
Indexed Flash Region (IFR)	NXP term for reserved flash regions with special purposes. Usually not directly programmable by the OEM.
Joint Test Action Group (JTAG)	An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a JTAG interface port to communicate with processors
JTAG	See *Joint Test Action Group*.
Non-Secure Processing Environment (NSPE)	Non-Secure Processing Environment
NSPE	See *Non-Secure Processing Environment*.
OEM	See *Original Equipment Manufacturer*.
Original Equipment Manufacturer (OEM)	Original Equipment Manufacturer, the device owner.
PKI	See *Public Key Infrastructure*.
Public Key Infrastructure (PKI)	Public Key Infrastructure
Requester	The Requester is the party initiating the protocol. For ADAC this is the host side.
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. See *Arm® Platform Security Architecture Security Model* [PSA-SM].
Root of Trust Public Key (ROTPK)	A public key programmed into immutable memory of a device.
RoT	See *Root of Trust*.
ROTPK	See *Root of Trust Public Key*.
SDA	See *Secure Debug Authenticator*.
SDM	See *Secure Debug Manager*.
Secure Debug Authenticator (SDA)	Component residing in the debug target that receives and verifies requests to unlock debug access.
Secure Debug Manager (SDM)	Component residing in the debug host that asks the Secure Debug Authenticator for debug access.
Secure Processing Environment (SPE)	Secure Processing Environment
Serial Wire Debug (SWD)	A low pin-count physical interface for JTAG debugging on ARM-processors.

Table 2 (continued)

Term	Meaning
Silicon Partner (SiP)	Silicon Partner
SiP	See *Silicon Partner*.
SPE	See *Secure Processing Environment*.
SWD	See *Serial Wire Debug*.
UDE	See *Unprivileged Debug Extension*.
Unprivileged Debug Extension (UDE)	Unprivileged Debug Extension

Potential for change

The contents of this specification are subject to change.

In particular, the following may change:

- Feature addition, modification, or removal
- Parameter addition, modification, or removal
- Numerical values, encodings, bit maps

Conventions

Typographical conventions

The typographical conventions are:

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

Numbers

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

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

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book is not expected to contain assembler code or pseudo code examples.

Any code examples are shown in a `monospace` font.

Current status and anticipated changes

First draft, major changes and re-writes to be expected.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title (Authenticated Debug Access Control).
- The number and issue (DEN 0101 1.0.1).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1 Debug Access Control Architecture

1.1 About the Architecture

This chapter describes the goals and scope of the ADAC architecture. It contains the following sections:

- About Platform Security Architecture
- Goals
- Scope

1.1.1 About Platform Security Architecture

This document is one of a set of resources provided by Arm that can help organisations 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.

1.1.2 Goals

Introducing security in debug is about making sure that only authorized people have access to select parts of firmware and hardware.

ADAC aims at making sure that debug capabilities do not become attack vectors. Debug security cannot be an afterthought when designing an SoC and the kind of debug solution needed is driven by the threat models for the device use case.

The ADAC architecture is designed to be flexible to meet varying vendor needs, adaptable to work with many different hardware and software components, and scalable from small embedded or IoT systems to complex server environments. At the same time, it strives to be simple and resilient against attack.

This requires:

- Strong authentication
- Partitioning firmware and hardware into fine-grained domains
- Enforcing debug limitations

ADAC needs to offer fine-grained access control of system resources, accessed through the debug port, across device lifecycle states.

This specification targets functional layers that sit above the physical debug link. As such, any security aspects of the physical layer, including confidentiality and integrity, are out of scope for this document.

1.1.3 Scope

The ADAC specification covers the following topics:

- Command protocol
- Debug authentication commands
- Certificate format
- Token format

The specification considers both device-level and application or software-level debug.

The specification does not cover enforcement of debug signals.

1.2 System Architecture

This chapter describes the overall architecture of ADAC. The contains the following sections:

- System architecture overview
- Target Architecture
- Protocol Architecture

1.2.1 System architecture overview

The high level ADAC system architecture is described by High level system block diagram.

It is composed of these two systems:

- *Debug target*: The system or subsystem containing the resources for which permission to debug is requested through the secure debug protocol.
- *Debug host*: The device that initiates the debug session. It drives the authentication sequence and provides credentials to the debug target.

The debug host and debug target are connected via a debug link. This is any interface from which the debug host can perform debug operations on the target. This can be debug probe hardware driving a wire protocol such as SWD or JTAG, self-hosted debug capability between cores in a multi-core device, the self-hosted debug capability of a single processor debugging itself, or any similar debug link.

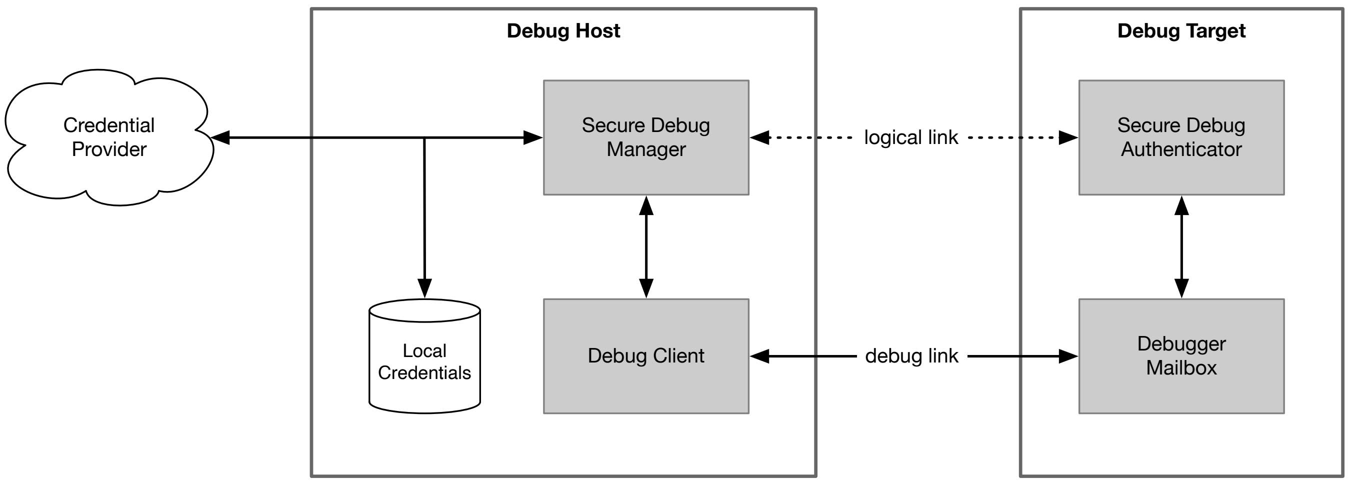


Figure 1 High level system block diagram

The following components are used to establish the communications channel over which authentication is performed.

- *Debug client*: The component that drives the debug link between host and target (e.g., debugger software).
- *Debugger mailbox*: Generic term for a communications endpoint, inside the debug target, where the Secure Debug Authenticator (see definition below) can receive commands. The key attribute is that a debugger mailbox is available even when debug access is otherwise restricted due to the device security lifecycle. However, there can be other temporal restrictions on when the Secure Debug Authenticator is available to respond to requests.

The system architecture avoids placing requirements on the debug link and communications channel, except to define the debugger mailbox as the endpoint through which commands and responses are transferred. This allows a common architecture to support both externally-hosted debug and self-hosted debug.

For externally hosted debug, the debug link is typically in the form of a debug probe connected to the host with a high-bandwidth protocol such as USB or Ethernet.

The two primary components that implement this architecture are:

- *Secure Debug Manager* (SDM): Initiates the logical communications link with the Secure Debug Authenticator on behalf of the debug client and manages the secure debug protocol. It receives credentials from either a local or remote credential provider, and passes those credentials to the Secure Debug Authenticator. The debug client and SDM can communicate to allow a user to choose requested permissions and credentials, or the selection of permissions and credentials can be fully automated.
- *Secure Debug Authenticator* (SDA): Accepts commands sent by the SDM via the debugger mailbox. It issues an authentication challenge and validates the credentials provided in response. Upon successful authentication, the SDA handles the hardware or software aspects of enabling debug access to target resources. In addition, upon request it can provide the SDM with information about the debug target. This information can be used for discovery and identification purposes, to pass to the credential provider, to present the user with data to make an informed decision when choosing credentials, or other purposes.

The SDM resides on the debug host, and the SDA resides on the debug target. The two components establish a logical communications link between themselves. This logical link is routed through the debug client, debug link, and debugger mailbox.

The SDA must be contained within a trusted domain. It can run from several possible execution contexts:

- Immutable Root of Trust (e.g., boot ROM)
- Mutable Root of Trust (e.g., updatable bootloader)
- Trusted runtime service

Each execution context has distinct properties and capabilities that can influence the debug authentication sequence.

The execution context of the SDA is not required to be in the same system or PE over which it controls access. For instance, an SDA can execute from a secure enclave and control access only for the system outside the enclave.

1.2.2 Target Architecture

Access Control

For discussion of the target architecture, these terms are defined:

- *Access control signals*: Logical signals that control debug access to hardware or software components, or modify the functionality of components during the time when debug access is enabled. The signals themselves are IMPLEMENTATION DEFINED and can be implemented in either hardware or software.
- *Access control domain*: The system or subsystem over which the SDA controls debug access via the access control signals. An access control domain can be hardware-defined, software-defined or a combination of both. It may only exist after a particular point in the boot process, such as in the case where the SDA is a trusted runtime service.

It is acceptable for a single SDA to control more than one access control domain. There can be more than one access control domain in a debug target, and thus more than one SDA. The method of selecting the access control domain for which the SDM is authenticating is IMPLEMENTATION DEFINED.

These are several examples of functions that access control signals can serve:

- Control debug access to a CPU or a security or privilege level within a CPU.
- Control access to a system memory bus and/or resources accessible through such a bus.
- Control the availability of certain cryptographic keys in a hardware cryptography accelerator.
- Force the hiding of secrets in the root of trust.

The access control domains and methods used by the SDA to apply access control signals within the domains are both IMPLEMENTATION DEFINED.

Target States

An access control domain can be in one of the following states:

- *Locked*: Default permissions as defined by the access control domain.
- *Unlocked*: This state is entered when any additional permission is granted.

Handling Reset

Note:

Whether this section applies to an access control domain that only exists after the system boots to a certain stage is determined by the system architecture of the debug target.

There are two types of reset that impact the ADAC architecture:

- Cold Reset. Often called a Power-On Reset.
- Warm Reset.

It is not possible to have a Cold reset without also having a Warm reset.

Access control signals must be reset to defined state, dependent on the device's security lifecycle state, on a Cold Reset. When in the Secured lifecycle state, a Cold reset should place the target in the Locked state.

If the target architecture allows, access control signals should remain unmodified through a Warm reset. This reduces the need for reauthenticating during target debug sessions. It can also allow for debug of the boot process. A target in the Unlocked state can temporarily disable debug access during boot following a Warm reset to ensure that execution of a boot ROM and/or other sensitive parts of the boot flow, such as secure boot, are protected.

1.2.3 Protocol Architecture

The ADAC specification defines the protocol used by the SDM to request debug access from the SDA. Several protocol layers are defined, as shown in Protocol layer stack.

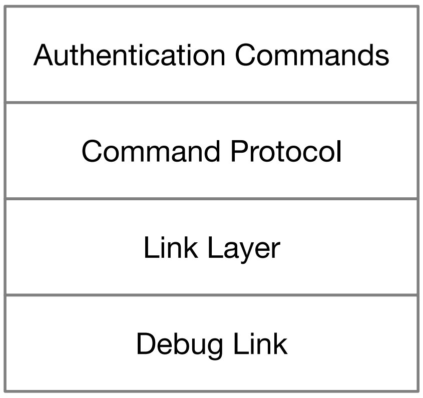


Figure 2 Protocol layer stack

Note that API layers are not included in this diagram.

The sections that follow describe each protocol layer in turn, from bottom up.

The bottom layer in the stack is the debug link, which was defined earlier.

Link Layer

The link layer sits atop the debug link and provides guarantees upon which the Command Protocol layer can be built.

Because the link layer is dependent upon the specific debug link, this specification does not require any one link layer. Several example link layers are documented in Link Layer.

Command Protocol

The purpose of the Command Protocol is to abstract the communication channel between host and target.

The Command Protocol provides a simple request/response mechanism suitable for implementing the Authentication Command Set atop arbitrary link layers.

Vendors can extend the usefulness of the debugger mailbox by designing vendor-specific command sets that reuse the Command Protocol.

The Command Protocol is documented in Command Protocol.

Authentication Command Set

The Authentication Command Set specifies a number of commands that implement ADAC. It defines these aspects of the protocol:

- The command sequence used to perform authentication.
- Status and error codes.
- Common data types.
- The set of supported data formats that implement the trust mechanism (see Security Model).

The principal elements of the trust mechanism are the ADAC Token and ADAC Certificate formats. Other authentication and trust mechanisms or formats, either standard extensions or proprietary, can also be used.

The Authentication Command Set is documented in Authentication Command Set.

1.3 Security Model

This chapter describes the security model for ADAC. The contains the following sections:

- About the Security Model
- Authentication
- Trust
- Constraints
- Examples

1.3.1 About the Security Model

Scope

Physical access to the target device being required for debug link operation, the main objective is for the target to securely authenticate the host. Authentication of the target by the host is not in scope for this document. Security of communications over the debug link, including confidentiality and integrity, are also not in scope for this document.

In order to simplify key management and improve security this specification focuses on asymmetric key cryptography.

Security Market Requirements

Use Cases

Use Cases in scope for ADAC are:

- Flash programming, e.g. for factory provisioning of code or credentials
- Loading code into RAM for immediate execution
- Memory upload/download
- Device configuration
- Extracting logs or any kind of data
- Recovery of bricked devices
- Tracing, both invasive and non-invasive
- Traditional run/stop debug during normal development

Stakeholders

Hardware and firmware vendors who want to protect device assets:

- Restrict peripheral setup and control
- Prevent firmware leakage or modifications
- Protect credentials from leakage or modification

Security Goals

Make sure the ADAC protocol cannot be abused to become a new attack vector, either by weakening its security or by setting the device to a non-working state.

Assets and Actors

The ADAC protocol is designed as a challenge/response exchange meant to set a bit vector defining debug permissions inside the target. The assets are the debug vector being negotiated.

The only actors considered in this analysis are the host and the target, as they are described in the protocol.

Trust boundaries

Host and target are considered trusted entities. Both host and target are required to hold credentials inside secure areas, e.g. a smart card for the host, a Root of Trust for the target. Securing host and target themselves is out of scope for this analysis, and heavily relies on their implementation.

The attack surface considered here is in the link between host and target.

Assumptions

Pre-provisioned keys

ADAC assumes that public keys of trusted authorities have been pre-provisioned on devices and are available to the ADAC target-side code. These keys are public but they should not be modifiable except in very specific cases, e.g. during a full update of the Root of Trust.

Error-free Link Layer

ADAC assumes the link layer used on debug ports is error-free. Unreliable link layers should be augmented with CRCs or similar error-correcting mechanisms to ensure proper delivery of messages without corruption.

Random Number Generation

The ADAC protocol relies on a one-time random challenge issued by the target. This assumes the target has access to enough entropy to generate enough unpredictable bits upon request.

1.3.2 Authentication

The default mechanism for authentication (see ADAC Token) relies on a challenge-response protocol. The challenge is used to protect against replay attacks.

Note:

The challenge vector can be:

- a cryptographically random value,
- a random value (low entropy),
- a combination of different elements that will make the challenge to be different when performing authentication on distinct devices and when performing multiple authentication on the same device. Those elements can be (non-exhaustive list): - device unique constant values (e.g. serial number), - non-repeatable values (e.g. monotonic counter, secure time), - device constant values with some entropy (e.g. MAC address, manufacturing date), - non-constant values (e.g. high-precision counter or clock).

  The randomness, non-malleability and unpredictability of the challenge vector is important to avoid the re-use of tokens for a given device or across devices. The options above are listed in a generic order of preference that might not apply to all cases.

The response to the challenge is a signed authentication token, also called the debug token in this specification. The key used to verify the signature must be trusted (see Trust).

1.3.3 Trust

In order to verify the signature of the authentication token, a (public) key is needed.

The simplest option has this key present on each device directly. This yields two extremes in terms of management options (or a combination of the two):

- Use the same key for all devices.

- Use a different key for each device.

A chain of certificates links the key used to sign to the authentication token to a set of one or more trusted anchors (roots of trust). Based on the vendor needs, the chain can be of arbitrary length, ending in a key directly linked to a programmed root authority (e.g. hash of public key(s) programmed into OTP).

Adding intermediate steps in the certificate chain adds some overhead to the authentication process in the form of extra verification operations and increased data size. However, intermediate certificates also limit the exposure of the most sensitive keys, allowing that those keys can be used less often and protected with added security (e.g., stored on offline/air-gapped systems or other physical protections).

This specification defines a certificate format (see ADAC Certificate) to build trust chains and offer the flexibility to deal with complex scenarios (see Constraints).

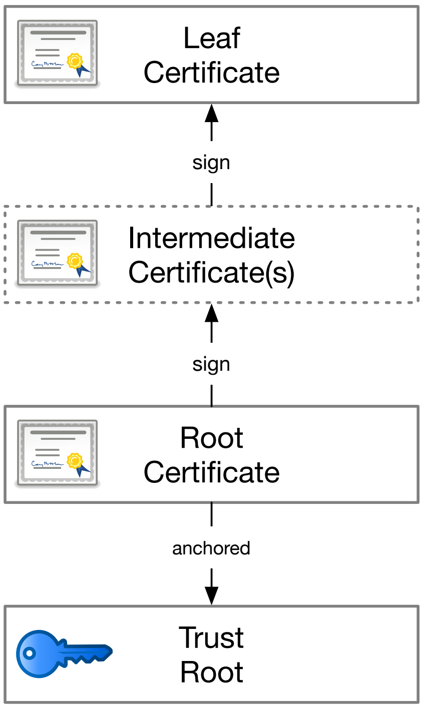


Figure 3 Example chain of trust.

The diagram in Example chain of trust. shows an example trust chain composed of a leaf certificate, zero or more intermediate certificates, ending in a root certificate. The certificate chain is anchored to the device's root of trust.

1.3.4 Constraints

Each certificate in the chain can add constraints to the authentication process in order to limit the scope of authentication and restrict permissions that its holder can unlock. This allows granting a specific set of privileges to a specific set of targets to either exercise or optionally to delegate further by issuing sub-certificates.

Two types of constraints are supported:

- Scope-limiting constraints. Described below.

- Permission-limiting constraints. See Permissions.

Scope-limiting constraints

Several scope-limiting features are included in each certificate:

- `lifecycle`: Limits to a specific lifecycle state.
- `soc_class`: Limits to specific vendor defined family of devices.
- `soc_id`: Limits to specific device.
- `oem_constraint`: Limits to devices with a matching static OEM constraint value.

Using certificate extensions it is possible to add other types of constraints (e.g. `target_identity` or `sw_partition_id`).

All scope-limiting constraints have a neutral value to indicate no further restriction. Values for these constraints in certificates need to be consistent both with the target configuration and with all other certificates in the certificate chain:

- The "expected value" of all the scope-limiting constraints listed above is derived from the target's configuration and state in an IMPLEMENTATION DEFINED manner.
- The "effective value" of a given constraint is the non-neutral value specified by the certificate chain.
- If two or more distinct non-neutral values are present in the certificate chain, a failure is triggered. In other words, only a single effective value is allowed.
- If a constraint's effective value does not match its expected value, the target will reject the authentication request.

Permissions

This specification supports two different permissions models:

- Logical permissions bits.
- Software partition permissions.

Permissions bitmap

For fine-grained access-control, this specification defines a standard mechanism to associate certificates in the chain with a bitmap of logical permissions.

The debug host requests access to a set of permissions via the authentication token. The effective permissions are computed by masking requested permissions with permission-limiting constraints from the certificate chain. This mechanism allows for certification authorities to restrict permissions of issued certificates.

The exact semantics for the permissions is an IMPLEMENTATION DEFINED combination of SoC-specific access control signals.

Logical permissions do not necessarily map 1:1 to debug access signals. The Secure Debug Authenticator implementation can apply an IMPLEMENTATION DEFINED mapping operation to convert logical effective permissions to the debug access signals programmed then into system control registers. This allows for compression of debug access signals into a simpler set of permissions, as well as to ensure a consistent security configuration.

A value of 1 for a logical permission bit means that access is granted, while a value of 0 means that access is denied.

Computing effective permissions

Definitions:

- `Perm_req`: Permissions vector requested by the debug host.
- `Perm_mask`: Mask of permissions allowed by the certificate chain to be enabled.
- `Perm_eff`: Final effective permissions after masking the requested permissions.
- `Soc_mask`: Static permissions mask value provided to the Secure Debug Authenticator.
- `Soc_value`: Static value used for permissions that, due to `Soc_mask`, are not allowed to be controlled via debug authentication.
- `Crt_count`: Number of certificates in the chain. The leaf certificate is at index 0, and the root certificate is at index `Crt_count - 1`.

`Soc_mask` and `Soc_value` are static values passed to the Secure Debug Authenticator from an IMPLEMENTATION DEFINED source. The intended use case is to allow the provisioning process of the target to set permanent restrictions on permissions requested by the host, and further, to set the values for those permissions that are restricted. These values can be programmed into an SoC's OTP memory or another trusted memory, fixed in hardware, set at software development time, or can simply be set to 0 if unavailable or not desired.

Steps to compute the effective permissions:

- The host requests a set of debug signals (`Perm_req`).
- The target combines the permission masks present in the certificates of the trust chain (`Perm_mask`):

```
let Perm_mask = ~0 // Initialize to all 1s.
for n in 0..(Crt_count - 1)
    let Perm_mask = Perm_mask & crt[n].permissions_mask
```

- Finally, the target computes the effective permissions (`Perm_eff`), merging with the SoC-programmed permission constraints (`Soc_mask` and `Soc_value`):

```
let Perm_eff = (Perm_req & Perm_mask & Soc_mask) | (Soc_value & ~Soc_mask)
```

The effective debug signals (`Sig_eff`) will then be used by the Secure Debug Authenticator to alter the debug configuration of the target system in an IMPLEMENTATION DEFINED manner.

Software partition permissions

Access to software partitions can also be requested in the authentication token, and restricted by the certificate chain.

Each software partition is identified by a unique ID. The authentication token includes a list of zero or more software partition IDs for which access is requested.

Certificates also include a list of zero or more software partition IDs. The rules for interpretation are as follows:

- If a partition ID appears in a certificate, then the certificate is declaring that debug access to that software partition can be granted if requested in the authentication token.
- If at least one software partition ID is listed anywhere in the certificate chain, then access to only those software partitions whose IDs are listed in the certificate chain can be granted.
- If no software partition IDs are are listed in the certificate chain, then debug access to software partitions is not constrained, and access to any partition whose ID is listed in the authentication token can be granted.

Other factors can influence whether permission to access a given software partition is granted. For instance, if the SoC-programmed permission constraints disallow any debugging of the SPE, then a request for debug access to an Application RoT partition must be disallowed. The interpretation of these additional factors and how they are applied (for instance, by hardware or software) is IMPLEMENTATION DEFINED.

The definition of software partition and the definition and size of software partition IDs is outside the scope of this specification.

1.3.5 Examples

The following scenarios illustrate some of the flexibility of the authentication mechanism combined with certificates:

- Manufacturing equipment with a device-class certificate (and matching key stored in a hardware security module) is able to authenticate itself to the devices on the production line to initialize them (flash, credentials, root of trust…).
- A developer uses a device-locked certificate with a local key to debug their application on the device.
- A technician connects diagnostics equipment to a device, the authentication token is generated in the cloud to unlock access and perform maintenance.

2 Specification

2.1 Common Elements

This chapter covers common definitions and conventions used throughout the ADAC specification. It contains the following sections:

- Conventions
- Version Numbers
- TLV Data Type
- Type ID Registry
- Key and Signature Types

2.1.1 Conventions

Data encoding conventions

The size of all outer-level aggregate data types must be 32-bit word aligned.

All multi-byte scalar values must be encoded in little-endian byte order.

Non-scalar, untyped, multi-byte arrays are encoded as little-endian byte arrays. As an example, take the value “DDCCBBAA8877665544332211”, written as a big-endian hex string. The most-significant byte, bits [95:88], has the value 0xDD. It is encoded as the byte sequence:

```
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xAA 0xBB 0xCC 0xDD ]
```

This table shows the same encoded value in several formats:

Table 3 Array encoding

Word Number	Bytes	32-bit Integer
0	[0x11 0x22 0x33 0x44]	0x44332211
1	[0x55 0x66 0x77 0x88]	0x88776655
2	[0xAA 0xBB 0xCC 0xDD]	0xDDCCBBAA

C language conventions

C structure definitions are used in this specification as a convenient syntax method for defining data types. However, the definitions should be considered pseudocode, as they can include non-conformant struct member definitions that are intended to better describe data sizes and relationships. In addition, all structures are defined as if packed (i.e., alignment is set to 1 byte), and include explicit padding to ensure the desired alignment of members.

C99 standard scalar types such as `uint32_t` are used for all integer values.

The following macros are used in definitions.

```
// Round a size *n* up to the nearest *r* multiple.
#define ROUND_UP(n, r) (((n) + (r) - 1) / (r) * (r))

// Round a size *n* up to the nearest 32-bit word.
#define ROUND_TO_WORD(n) (ROUND_UP((n), sizeof(uint32_t)))
```

2.1.2 Version Numbers

Version numbers are used throughout the specification in order to allow for future changes while remaining backwards compatible. The authentication protocol itself has a version. And each major aggregate data type has a version number as its first member.

Version numbers consist of these two components:

- *Major version*: Must only be incremented when the data type is entirely redefined.
- *Minor version*: Must be incremented due to added or changed data type members, where the majority of the data type remains compatible.

Version numbers are encoded as a sequence of two 8-bit integers, with the major version first and minor version following. A C structure definition follows.

```
struct adac_version {
    uint8_t major;
    uint8_t minor;
};
```

For example, version 1.2 is represented by the byte sequence `[ 0x01 0x02 ]`.

Version number counting starts at version 1.0.

2.1.3 TLV Data Type

A simple Type-Length-Value (TLV) data type is used in multiple places throughout the specification.

Each value consists of a 32-bit header with type ID and length, followed by the value data. The size of the entire TLV instance must be rounded up to the nearest 32-bit word.

Each TLV instance consists of the following fields:

Table 4 TLV Fields

Name	Bytes	Description
(reserved)	2	Reserved for future use. Must be set to a value of `0`.
Type ID	2	Unique identifier for the value type.
Length	4	Length of value in bytes. Does not include the size of any required padding.
Value	*n*	Value data. Must be padded with `0` to align on a 32-bit boundary

The C type definition for a TLV instance follows.

```
struct adac_tlv {
    uint16_t _reserved;
    uint16_t type_id;
    uint32_t length_in_bytes;
    uint8_t value[ROUND_TO_WORD(length_in_bytes)];
};
```

In standard usage within other data types, multiple TLV instances are placed back to back in a variable-length array. This construction is called a “TLV sequence”. Because the size of every TLV instance is rounded up to be 32-bit aligned, all TLV instances naturally start on 32-bit boundaries. The total size of the TLV sequence is specified by another member of the parent data type.

TLV sequence elements should be ordered by type ID. More than one instance of a given type ID is allowed within a TLV sequence.

Nested TLV types are not used in this specification.

2.1.4 Type ID Registry

The type ID for values in TLV instances is a 16-bit value. Any type ID with bit 15 set is vendor-specific. This leaves room for 32768 possible standard type IDs. However, related type IDs are grouped into consecutive ranges, so the ID space is sparsely populated.

The following table contains the complete list of type IDs used for TLV instances in this specification.

Table 5 Type ID Registry

ID	Name	Bytes	Description
0x0000	`null_type`	0	Indicates “no data”
0x0001	`adac_auth_version`	2	Major and minor versions for ADAC. See *TODO*
0x0002	`vendor_id`	2	Vendor JEP106 ID
0x0003	`soc_class`	4	SoC class
0x0004	`soc_id`	16	SoC unique identifier
0x0005	`target_identity`	*n*	Cryptographic identity for the target
0x0006	`hw_permissions_fixed`	16	Value of hardware-fixed permissions
0x0007	`hw_permissions_mask`	16	Mask of permission bits allowed to be modified
0x0008	`psa_lifecycle`	2	Current lifecycle state
0x0009	`sw_partition_id`	*n*	Persistent, unique ID for a software partition
0x000A	`sda_id`	1	Unique ID for the SDA on the target
0x000B	`sda_version`	*n*	Vendor-specific SDA version
0x000C	`effective_permissions`	16	Computed permissions vector
0x0100	`token_formats`	2 * *n*	Array of supported debug token formats
0x0101	`cert_formats`	2 * *n*	Array of token and certificate formats
0x0102	`cryptosystems`	1 * *n*	List of *TODO* supported algorithms plus key sizes
0x0200	`token_adac`	*n*	ADAC token format
0x0201	`cert_adac`	*n*	ADAC certificate format
0x0202	`rot_meta`	*n*	Platform-specific metadata about the root of trust
0x8000			Start of vendor-specific value type IDs

Note:

Not all type IDs are accepted in all situations. The specification of each data type that uses TLV will contain a list of accepted type IDs.

Detailed information of each type ID follows.

- `adac_auth_version`: The version number for the authentication protocol command set. Currently defined as version 1.0.
- `cert_adac`: The ADAC Certificate format.
- `cert_formats`: Array of 16-bit type IDs indicating which certificate formats are supported by the debug target. In the current version of the specification, this may include `cert_adac` and `rot_meta`, plus

any vendor-specific formats.

- `cryptosystems`: Array of 8-bit cryptosystem IDs indicating those cryptosystems supported by the debug target.

- `effective_permissions`: Permissions vector resulting from a successful authentication command sequence. Note that this is the SDA's view of effective permissions; it may not match actual permissions allowed by the hardware in cases where the SDA is unable to determine whether a permission will be honored.

- `hw_permissions_fixed`: Bitfield setting the fixed value of any permissions bits whose corresponding bit in `hw_permissions_mask` is cleared.

- `hw_permissions_mask`: Hardware-defined permissions mask. When a permission bit is set to 0 in this field, it indicates that the hardware disallows modification of that permission via the authentication protocol—the permission is always fixed to the value of the corresponding bit in `hw_permissions_fixed`.

- `null_type`: The ID 0x0000 is reserved and is used to represent "no data" or list termination in special cases.

- `psa_lifecycle`: Represents the current lifecycle state of the PSA RoT. The state is represented by an integer that is divided to convey a major state and a minor state. A major state is defined by [PSA-SM]. A minor state is optional, and is IMPLEMENTATION DEFINED. The PSA security lifecycle state and implementation state are encoded in Trusted Firmware-M (<https://www.trustedfirmware.org>) as follows:
  - version[15:8] – PSA security lifecycle state (see values below)
  - version[7:0] – IMPLEMENTATION DEFINED state.

Table 6 PSA security lifecycle state values

Name	Value
`PSA_LIFECYCLE_UNKNOWN`	`0x0000`
`PSA_LIFECYCLE_ASSEMBLY_AND_TEST`	`0x1000`
`PSA_LIFECYCLE_PSA_ROT_PROVISIONING`	`0x2000`
`PSA_LIFECYCLE_SECURED`	`0x3000`
`PSA_LIFECYCLE_NON_PSA_ROT_DEBUG`	`0x4000`
`PSA_LIFECYCLE_RECOVERABLE_PSA_ROT_DEBUG`	`0x5000`
`PSA_LIFECYCLE_DECOMMISSIONED`	`0x6000`

- `rot_meta`: Optional vendor-defined data required by a Secure Debug Authenticator to verify the provided public root key matches the hardware Root of Trust Public Key (ROTPK).

- `sda_id`: 1-byte identifier for the Secure Debug Authenticator, unique within the scope of the target device. Used to distinguish between SDAs on a target with multiple logical or physical SDAs. The value is recommended to be a simple index, although if appropriate it can be a more complex value with target-specific meaning.

- `sda_version`: Version number of the SDA implementation represented as a UTF-8 encoded string with no terminating null byte. The version number should match the software version of the PSA Root of Trust within which the SDA is running.

- `soc_class`: Vendor-unique identifier for the SoC part number and revision. The layout and meaning of any individual fields within `soc_class` is the responsibility of the vendor.

- `soc_id`: 128-bit identifier for the SoC. For a given root of trust, this value should be unique. This value is not sensitive. It is tied to the hardware and does not change for the life of the physical SoC. Often, the ID is a serial number composed of die and wafer coordinates plus a random number programmed into OTP memory by the silicon vendor. When combined with `vendor_id` and `soc_class`, the result becomes globally unique.

- `sw_partition_id`: This value uniquely identifies a software partition. The length and semantics of a partition ID value are not determined by this specification.

- `target_identity`: Cryptographic identity for the target. The length of this value depends upon the cryptosystem used for its construction. Might not be available in all circumstances. For instance, a boot ROM might not have access to the information required to construct this value.

- `token_adac`: The ADAC Token format.

- `token_formats`: Array of 16-bit type IDs indicating which token formats are supported by the debug target.

- `vendor_id`: JEDEC JEP106 vendor ID. Bits [6:0] hold the "Identity Code" value; bits [15:7] contain the count of 0x7F Continuation Codes. For example, Arm's `vendor_id` is 0x023B.

2.1.5 Key and Signature Types

The specification currently supports the following key types:

- ECDSA P-256 (see [P-256 Curve])
- ECDSA P-521 (see [P-521 Curve])
- RSA 3072 (see [RSA 3072-bit keys])
- RSA 4096 (see [RSA 4096-bit keys])
- Ed25519 (see [Ed25519 Curve])
- Ed448 (see [Ed448 Curve])
- SM2 (see [SM2])
- CMAC (see [CMAC with AES])

- HMAC (see [HMAC with SHA-256])

The specification currently supports the following signature types:

- ECDSA P-256 with SHA-256
- ECDSA P-521 with SHA-512
- RSA 3072 with SHA-256
- RSA 4096 with SHA-256
- Ed25519 with SHA-512
- Ed448 with SHAKE256
- SM2 with SM3
- CMAC with AES
- HMAC with SHA-256

Currently a given key type only supports one signature algorithm.

The list of accepted cryptosystem combinations and the unique constants for identifying each follows.

A cryptosystem ID is an 8-bit value. Vendor-defined cryptosystems are allowed by setting the MSB (bit 7) of the ID.

Table 7 Supported cryptosystems

ID	Name	Public Key	Signature Algorithm
0x01	`ECDSA_P256_SHA256`	Elliptic Curve P-256	ECDSA with SHA-256
0x02	`ECDSA_P521_SHA512`	Elliptic Curve P-521	ECDSA with SHA-512
0x03	`RSA_3072_SHA256`	RSA (3072-bit key)	RSA-PSS with SHA-256
0x04	`RSA_4096_SHA256`	RSA (4096-bit key)	RSA-PSS with SHA-256
0x05	`ED_25519_SHA512`	Ed25519	EdDSA with SHA-512
0x06	`ED_448_SHAKE256`	Ed448	EdDSA with SHAKE256
0x07	`SM_SM2_SM3`	SM2	SM2 with SM3
0x08	`CMAC_AES`	Nonce	CMAC with AES
0x09	`HMAC_SHA256`	Nonce	HMAC with SHA-256
0x80	Start of vendor-defined cryptosystems		

The IDs listed above are used in several places:

- The `cryptosystems` value (see above).
- Signature algorithm ID in the ADAC Token header.
- Key and signature algorithm ID in the ADAC Certificate header.

Cryptosystem Support

Debug targets are expected to support only one or a small number of related cryptosystems. For instance, a debug target might support only `ECDSA_P256_SHA256`, or could support both `RSA_3072_SHA256` and `RSA_4096_SHA256`.

It is likely that the debug host supports more cryptosystems than the target. On the other hand, vendor-specific implementations of the Secure Debug Manager can choose to implement only those cryptosystems known to be supported by that vendor's target-side implementation.

2.2 Command Protocol

This chapter specifies the generic high level command protocol. It contains the following sections:

- About the command protocol
- Protocol state machine
- Packets
- Request
- Response
- Error Handling

2.2.1 About the command protocol

The Debug Mailbox functionality can and has been implemented in many different ways. In order to abstract this implementation aspect, this document defines a command protocol. The link layer and wire protocol are left to be implementation specific.

In order to support most existing debug mailbox solutions and not create additional requirements for future ones, the goal is for the command protocol to be extremely simple. This facilitates simple target-side software, which is important for constrained systems and execution environments such as a boot ROM. Critically, it also makes reasoning about the security aspects of an implementation much easier.

Key attributes of the command protocol:

- Variable size, word aligned packets.
- Simple request/response model.

Non-requirements:

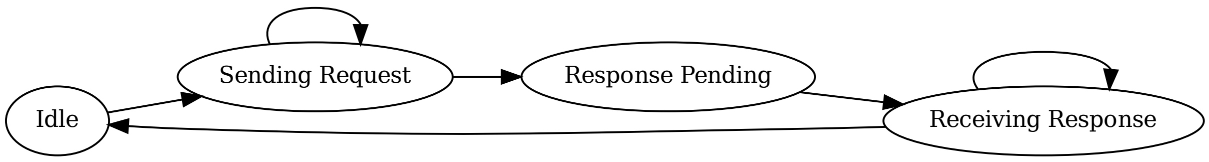
- Error correction.
- Flow control.
- Multiple in-flight commands.
- Out of order packets.
- Fixed size packets.

The link layer is assumed to provide error correction and flow control where necessary. Some link layers additionally provide their own packetization method.

All packets are 32-bit word aligned. This easily supports debugger mailboxes that transfer either byte or word sized data. It is assumed that commands can easily arrange for word aligned parameters.

2.2.2 Protocol state machine

The following diagram depicts the state machine for the command protocol.



The states are labeled from the point of view of the debug host. For the debug target, the sending and receiving roles are reversed.

In this protocol, the debug host is always the Requester and initiator of commands.

2.2.3 Packets

Request

A request consists of a two-word header word containing the command ID and request data length in bytes, followed by the optional request data. The complete request size must be word aligned; thus, the two least-significant bits of `data_count` must be zero.

A request must only be sent from the Idle protocol state.

The high bit (bit 15) of the command ID indicates a vendor-specific command.

Table 8 Request bytes

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	`(0)`	`(0)`	`command[7:0]`	`command[15:8]`
1	`data_count[7:0]`	`data_count[15:8]`	`data_count[23:16]`	`data_count[31:24]`
2	`data`	…	…	…

The C structure definition for a request is as follows:

```
struct request_packet {
    uint16_t _reserved;
    uint16_t command;
    uint32_t data_count;
    uint32_t data[];
};
```

Response

Similar to a request, a response packet has a two-word header containing the command status and response data byte count. Following the header is the optional response data. As with requests, the total size must be word aligned, and the two least-significant bits of `data_count` must be zero.

A response packet is always a reply to the most recent request. Because the protocol does not allow for out of order or asynchronous commands, there is no need to include a copy of the command ID in the header or a sequence number.

The response to a command must be sent only in the Response Pending protocol state by the receiver of the request. No intervening packets, sent by either endpoint, are allowed between the request and response. Once the response is received, the protocol transitions back to the Idle state, and another request can be sent.

Table 9 Response bytes

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	`(0)`	`(0)`	`status[7:0]`	`status[15:8]`
1	`data_count[7:0]`	`data_count[15:8]`	`data_count[23:16]`	`data_count[31:24]`
2	`data`	`...`	`...`	`...`

The C structure definition for a response is as follows:

```
struct response_packet {
    uint16_t _reserved;
    uint16_t status;
    uint32_t data_count;
    uint32_t data[];
};
```

2.2.4 Error Handling

Error handling capabilities of the command protocol are intentionally limited. It is assumed that either a method specific to the link layer or a system reset can be used to restore communications channel functionality in the event of an irrecoverable error.

The status value of 0x7FFF is reserved for unrecognized commands. If a request with an unrecognized command ID is received by the target, it must reply with a response packet consisting of a status value of 0x7FFF and no data words.

As a byte sequence, the error packet is:

```
[ 0x00 0x00 0xFF 0x7F 0x00 0x00 0x00 0x00 ]
```

2.3 Authentication Command Set

This chapter specifies the debug authentication commands and their parameters. It contains the following sections:

2.3.1 About the authentication commands

ADAC defines the following set of commands for performing debug authentication. These commands are defined as a layer building on the Command Protocol.

All commands below must be implemented by the debug target. However, some commands can return a status code indicating they are unsupported on the target or in the execution environment. This is documented per command.

Detailed specifications for each command follow in this chapter.

Table 10 Command Set

ID	Constant	Command	Description
0x0001	`ADAC_DISCOVERY_CMD`	Discovery	Query target properties
0x0002	`ADAC_AUTH_START_CMD`	Authentication Start	Initiate authentication sequence; receive challenge
0x0003	`ADAC_AUTH_RESPONSE_CMD`	Authentication Response	Send authentication data
0x0004	`ADAC_CLOSE_SESSION_CMD`	Close Session	Terminate command session
0x0005	`ADAC_LOCK_DEBUG_CMD`	Lock Debug	Lock debug access; restore system to locked state

Additional command sets can be defined in further specifications.

Vendor-specific commands can be added by the integrator. All vendor-specific commands must have bit 15 set in the command ID.

Currently defined versions of the ADAC protocol are as follows. The version is reported via the adac_auth_version value returned by the Discovery command.

Table 11 Protocol Version

Version	Description
1.0	Initial version

Status codes

The following table lists the complete set of status codes returned by the authentication commands.

Table 12 Status Codes

Status	Constant	Description
0x0000	`ADAC_SUCCESS`	The command has succeeded without error
0x0001	`ADAC_FAILURE`	The command has failed
0x0002	`ADAC_NEED_MORE_DATA`	More data is required to complete the authentication
0x0003	`ADAC_UNSUPPORTED`	The command is not supported by the target
0x7FFF	`ADAC_INVALID_COMMAND`	The command ID is unrecognized

The status code 0x7FFF is special in that it is not returned by a specific command but instead indicates an unrecognized command ID. See the Error Handling section of the command protocol for more information.

Authentication response

A complete authentication response consists of a sequence of separate cryptographically signed data structures including one or more certificates and the debug token. It can additionally include vendor-specific credentials or cryptographic material required for the target to verify the certificate chain or debug token.

The certificate chain is required to be provided in order from root to leaf.

Response fragments

A response fragment is defined as one of the data structures composing the authentication response. The primary classes of response fragment are the debug token and certificate. Each class of response fragment has a set of valid formats in which it can be represented. Every supported response fragment format has a unique type ID.

The debug target is not required to support all possible response fragment formats, and indeed is expected to only support a small subset. The debug host can use the Discovery command to query the target for supported response fragment formats. This process is intended to be used as additional validation rather than for protocol negotiation.

Type IDs for accepted response fragment formats are listed in the following table.

Table 13 Type IDs

Value	Name	Description
0x0200	`token_adac`	ADAC Token format
0x0201	`cert_adac`	ADAC Certificate format
0x0203	`rot_meta`	Vendor-specific RoT metadata

Example authentication response

As an example, an authentication response can be constructed from this sequence of response fragments:

1. ADAC Certificate: Root certificate.
2. ADAC Certificate: Leaf certificate.
3. ADAC Token

The following diagram shows the relationships between the response fragments in this example.

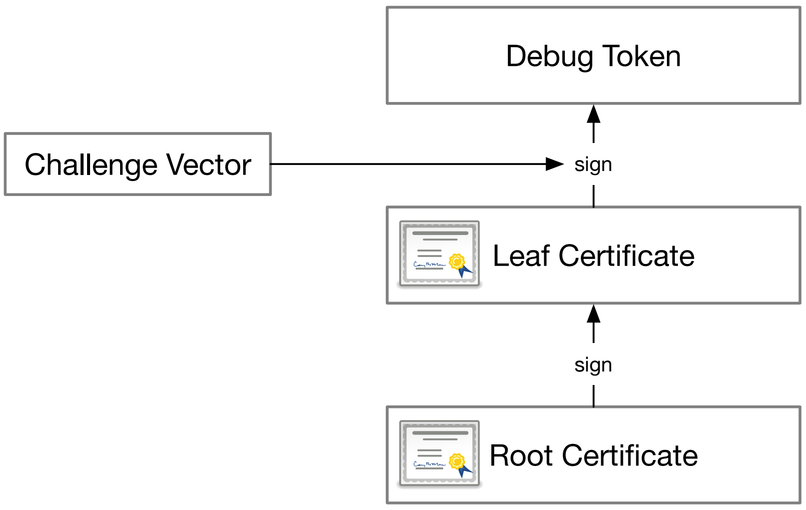


Figure 4 Example Authentication Response

The number of certificates used is a decision made by the customer to trade off security versus processing time and management complexity. This example uses two certificates; three is also common. Use of a single certificate is possible but not recommended.

As can be seen, the root certificate signs the leaf certificate. The leaf certificate then signs the debug token and challenge vector.

Note that the root certificate is tied to the target's hardware root of trust. Normally the root certificate contains the target's ROTPK.

Authentication command sequence

The process of enabling debug access to system resources is accomplished by sending a sequence of the commands defined in this chapter. Only some of the commands defined in this chapter are required as part of the authentication sequence; the others are optionally used for gathering information about the target or requesting a change in system state that does not require authentication.

The required sequence of commands for authentication is as follows.

1. Authentication Start: Host requests challenge from target.
2. Authentication Response (one or more): Host sends certificate chain (possibly other response fragments) and debug token to target.

The number of Authentication Response commands sent is determined by the number of response fragments from which the complete authentication response is constructed.

If Authentication Response returns the `ADAC_NEED_MORE_DATA` status code, then the target expects another Authentication Response command to be sent in order to continue or complete the authentication sequence.

Multiple authentication sequences are supported by the specification, but whether this is allowed is IMPLEMENTATION DEFINED. This can be used for various purposes, such as to unlock more than one access control domain, potentially with multiple roots of trust, to gain access to more than one software partition, or other purposes.

More than one authentication sequence in parallel is not allowed. This means that once an authentication sequence is started, no intervening commands are allowed to be issued until the sequence completes successfully or fails due to an error.

After all intended authentication sequences are complete, the Close Session command must be sent in order terminate communications.

Note:

The benefits of issuing one command per response fragment are two-fold:

1. Reduced target memory requirements by supporting incremental processing.
2. Allows for the possibility of early error termination.

2.3.2 Authentication Commands

Discovery

The debug host requests information about the debug target using with this command.

The debug host can optionally include a list of type IDs for values requested from the target. The requested ID list is discretionary; the target can reply with any set of values that is equal to or a super set of the requested values, excluding any values unavailable on the target or not recognized by the target.

If a requested ID list is not included, the target must reply with all available values.

Use of this command is optional and is not part of the required authentication command sequence. More than one Discovery command is allowed to be sent by the host.

Request

Command ID	ADAC_DISCOVERY_CMD (0x0001)
Request Data	Array of requested type IDs.

The request data is an optional array of requested type IDs, each a 16-bit half-word.

If the number of requested IDs is not even, then an extra null_type ID with value 0x0000 is appended to round up to the request data length to the next 32-bit word as required by the command protocol. If the null_type ID is present in the requested ID array at any position other than the last, it terminates processing of the array early.

The request sequence should be ordered by increasing ID value. If it is not, some values may be omitted from the results.

Response

Response Data	TLV sequence

Possible status codes:

Table 16 Discovery status codes

Status Code	Meaning
ADAC_SUCCESS (0x0000)	A valid discovery response follows
ADAC_FAILURE (0x0001)	Discovery failed

The response consists of a TLV sequence. The total size of the TLV sequence is indicated by the response packet header's data_count field. This provides a simple mechanism for managing variable length values. It also allows vendors to extend the response with custom information if needed.

Because this command is not critical to security of the overall protocol, and all response processing happens on the host, the required parsing of TLVs is not a concern in regards to target vulnerabilities.

Possible response data type IDs:

Table 17 Discovery response data type IDs

Value	Name
0x0001	adac_auth_version
0x0002	vendor_id
0x0003	soc_class
0x0004	soc_id
0x0005	target_identity
0x0006	hw_permissions_fixed

Table 17 (continued)

Value	Name
0x0007	`hw_permissions_mask`
0x0008	`psa_lifecycle`
0x000A	`sda_id`
0x0100	`token_formats`
0x0101	`cert_formats`
0x0102	`cryptosystems`

Vendor extension types are also allowed. The response sequence must be ordered by increasing ID value.

Example

```
// @+00 (12 bytes) psa_auth_version: 1.0
0x00, 0x00, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
// @+12 (12 bytes) vendor_id: {0x04, 0x3B} => 0x023B ("ARM Ltd.")
0x00, 0x00, 0x02, 0x00, 0x02, 0x00, 0x00, 0x00, 0x04, 0x3B, 0x00, 0x00,
// @+24 (12 bytes) psa_lifecycle: PSA_LIFECYCLE_SECURED
0x00, 0x00, 0x08, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x30, 0x00, 0x00,
// @+36 (12 bytes) token_formats: [{0x00, 0x02} (token_psa_debug)]
0x00, 0x00, 0x00, 0x01, 0x02, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
// @+48 (12 bytes) cert_formats: [{0x01, 0x02} (cert_psa_debug)]
0x00, 0x00, 0x01, 0x01, 0x02, 0x00, 0x00, 0x00, 0x01, 0x02, 0x00, 0x00,
// @+60 (12) bytes) cryptosystems: [ ECDSA_P256_SHA256 ]
0x00, 0x00, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
```

Authentication Start

The debug host sends this commands to start the authentication sequence. Its primary purpose is for the target to provide a random 256-bit challenge vector used to prevent replay attacks.

Request

Command ID	`ADAC_AUTH_START_CMD` (0x0002)
Request Data	None

Response

Response Data	`adac_auth_challenge` struct

Possible status codes:

Table 20 Authentication Start response status codes

Status Code	Meaning
ADAC_SUCCESS (0x0000)	A valid challenge structure follows
ADAC_FAILURE (0x0001)	Target is unable to send a challenge

The response data consists of the following versioned structure.

```
struct adac_auth_challenge_v1_0 {
    struct adac_version format_version;
    uint16_t _reserved;
    uint8_t challenge_vector[32];
};
```

Note that the challenge_vector field is encoded as a multi-byte array, as described under Data encoding conventions.

Currently defined versions of this structure are as follows.

Table 21 Authentication Challenge Version

Version	Description
1.0	Initial version

Authentication Response

This command is used to provide the debug token and additional credentials as part of a complete authentication response to the target. One or more Authentication Response commands must occur to form the complete authentication response

This command will return an ADAC_NEED_MORE_DATA status code to indicate that further data is required to complete the authentication and thus further Authentication Response must be sent. This sequence will continue until all required data has been provided to allow the target to validate the trust chain and debug token.

The target validates the provided debug token. If validation fails, the ADAC_FAILURE status is returned.

Request

Command ID	ADAC_AUTH_RESPONSE_CMD (0x0003)
Request Data	adac_auth_fragment_header plus response fragment

The data for Authentication Response request phase consists of a 32-bit header specifying the type ID of the included response fragment followed by the entire response fragment for the debug token.

The C definition of the response is as follows:

```
struct adac_auth_fragment_header {
    uint16_t fragment_type_id;
    uint16_t _reserved;
};

struct adac_auth_fragment {
    struct adac_auth_fragment_header header;
    uint8_t data[];
};
```

Response

Response Data	TLV sequence

Possible status codes:

Table 24 Authentication Response status codes

Status Code	Meaning
`ADAC_SUCCESS` (0x0000)	Authentication has succeeded. The authentication sequence is complete.
`ADAC_FAILURE` (0x0001)	Authentication failed
`ADAC_NEED_MORE_DATA` (0x0002)	More data is required to complete the authentication sequence

The response consists of a TLV sequence. The total size of the TLV sequence is indicated by the response packet header's `data_count` field.

The target can optionally return data pertinent to the authentication status, either success or failure. The primary use case is to allow the target to return the effective permissions resulting from a successful authentication. Vendors can extend the response with custom information specific to their use cases and platform if needed.

Possible response data type IDs:

Table 25 Authentication Response response data type IDs

Value	Name
0x0009	`sw_partition_id`
0x000C	`effective_permissions`

Vendor extension types are also allowed. The response sequence must be ordered by increasing ID value.

Close Session

This command requests that the communications session between the Secure Debug Manager and the Secure Debug Authenticator be terminated.

Depending on the execution environment of the Secure Debug Authenticator, closing the session can cause system boot to continue. Any means of resuming execution is acceptable, including performing a system reset. Note that if a system reset is used to resume execution, it should be performed after sending the response. Implementations should take link layer specifics into account to ensure a good likelihood of the response being delivered successfully.

Request

Command ID	`ADAC_CLOSE_SESSION_CMD` (0x0004)
Request Data	None

Response

Response Data	None

Possible status codes:

Table 28 Authentication Close Session status codes

Status Code	Meaning
`ADAC_SUCCESS` (0x0000)	The communication session was closed.

Lock Debug

The Lock Debug command restores the the device's debug access controls to the Locked state, given its current lifecycle state.

Not all targets will have the capability to lock debug access after it is unlocked without going through a Reset cycle (usually a Cold reset).

Request

Command ID	`ADAC_LOCK_DEBUG_CMD` (0x0005)
Request Data	None

Response

Response Data	None

Possible status codes:

Table 31 Authentication Lock Debug status codes

Status Code	Meaning
`ADAC_SUCCESS` (0x0000)	Debug access is now locked
`ADAC_FAILURE` (0x0001)	Debug access could not be locked because of the target configuration, runtime state, or other conditions
`ADAC_UNSUPPORTED` (0x0004)	Locking debug access is not supported by the target

The `ADAC_FAILURE` status should only be returned if the target does support restoring debug access locks at runtime in one or more configurations. If the target never supports the operation, then `ADAC_UNSUPPORTED` must be returned.

2.4 ADAC Token

This chapter specifies the structure of the binary debug authentication token. It contains the following sections:

- About the ADAC Token
- Format
- Extensions
- Allowed Extension Types
- Rules

2.4.1 About the ADAC Token

The ADAC Token is part of the authentication mechanism defined for ADAC. A token is sent by the debug host in response to a challenge vector (sometimes called nonce) sent by the target, and is cryptographically linked to the challenge vector and Root of Trust.

The ADAC Token also contains debug access permissions requested by the debug host.

A proprietary binary token format is used in order simplify requirements for parsing.

2.4.2 Format

The components of a ADAC Token are the following:

- *Header*: The header contains all mandatory fields and the length of extensions in bytes. The size of the header remains constant for different cryptosystems.
- *Extensions hash*: Hash of Extensions sequence. The algorithm and length depend on the Signature Type field in Header. Value is all zeros if extensions length is zero.
- *Signature*: The signature is performed over Header and Extension Hash parts of the Token as well as the challenge sent by the target.
- *Extensions*: TLV sequence of non-mandatory and vendor-specific fields.

Mandatory fields contained in the header:

- `format_version`: See Version.
- `signature_type`: Cryptosystem ID specifying the algorithm used to generate the token's signature, as well as the extensions hash.
- `requested_permissions`: Bitfield of requested debug permissions. A set bit indicates a request for the permission corresponding to that bit to be granted.

The data layout for the token header is shown in the following table.

Table 32 Token Header

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	`format_version.major`	`format_version.minor`	`signature_type`	`(0)`
1	`extensions_bytes`			
2	`requested_permissions[31:0]`			
3	`requested_permisions[63:32]`			
4	`requested_permisions[95:64]`			
5	`requested_permisions[127:96]`			

The following C structures describe the token header.

```
struct adac_debug_auth_token_header_v1_0 {
    struct adac_version format_version;
    uint8_t signature_type;
    uint8_t _reserved;
    uint32_t extensions_bytes;
    uint8_t requested_permissions[16];
};

struct adac_debug_token_v1_0 {
    struct adac_debug_auth_token_header_v1_0 header;
    uint8_t extensions_hash[HASH_LENGTH];
    uint8_t signature[SIGNATURE_LENGTH];
    // array of variable-length adac_tlv structs
    uint8_t extension_data[ROUND_TO_WORDS(header.extensions_bytes)];
};
```

Currently defined versions of the `adac_debug_token_v1_0` structure are as follows.

Table 33 Debug Token Version

Version	Description
1.0	Initial version

2.4.3 Extensions

The extensions for a ADAC Token are structured as a TLV sequence. In the `adac_debug_token_v1_0` struct, the extensions TLV sequence is placed in the `extensions_data` member.

The size of the extensions data is specified in the certificate header `extensions_bytes` member as a number of bytes.

Allowed Extension Types

The following table lists those Type IDs that are accepted in the extensions data for a ADAC Certificate. Any extension value with a Type ID not included within this list will be ignored.

Table 34 Type IDs

Type ID	Name	Description
0x0005	`target_identity`	Target identity
0x0009	`sw_partition_id`	Software partition ID

All vendor-specific type IDs are allowed.

The sw_partition_id extension specifies a software partition to which access is granted. This extension value can be included more than once, in which case access is granted to all listed software partitions.

2.4.4 Rules

Construction

The hash of the token extensions is computed as follows, using the hash algorithm identified by the `header.signature_type` cryptosystem.:

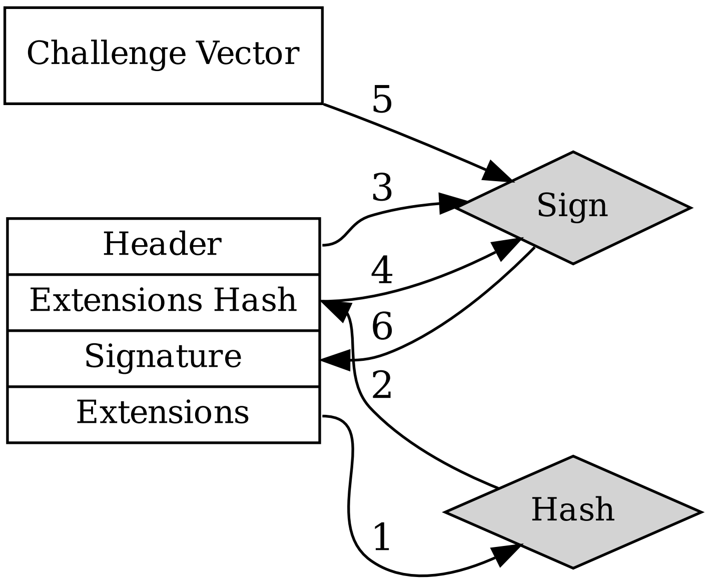
```
extensions_hash = Hash(extensions_data)
```

The signature over the token is computed as follows. The signature algorithm used is specified by the `header.signature_type` cryptosystem.:

```
signature = Sign(leaf_cert.private_key, header || extensions_hash || challenge_vector)
```

In the above expression, the symbol `leaf_cert` refers to the leaf certificate that signs the token.

The following diagram shows the inputs to the token signature algorithm.



Validation

The `header.format_version` member can be used to select an appropriate struct definition for parsing the entire header.

These members of the `adac_debug_token_v1_0` struct must be validated before any further processing of the certificate is performed:

- `header.format_version`
- `header.signature_type`

2.5 ADAC Certificate

This chapter specifies the structure of binary certificates used in ADAC. It contains the following sections:

- About the ADAC Certificate
- Format
- Extensions
- Allowed Extension Types
- Rules

2.5.1 About the ADAC Certificate

The high complexity of X.509v3 certificates, in addition to being costly (effort, code size, RAM, etc.), has been the source of bugs and security issues.

Due to the low bandwith of some of the underlying transports and the potential resource constraints of the target, this specification recommends the use of the alternative, purpose-built certificate format described in this chapter.

An ADAC Certificate is an element of the chain of trust. It also contains a set of optional constraints applied to debug authentication and debug permissions.

2.5.2 Format

The components of an ADAC Certificate are the following:

- *Header*: The header contains all fields and the length of the extensions in bytes. The size of the header remains constant for all cryptosystems.
- *Extensions hash*: Hash over extensions data. Algorithm and length depend on Signature Type field in Header. Value is all zeros if extensions length is zero.
- *Public key*: Content and length depend on Key Type field in Header.
- *Signature*: The signature is performed over Header, Extension Hash, and Public Key.
- *Extensions*: TLV sequence of non-mandatory and vendor-specific fields.

Fields contained in the header, all mandatory:

- `format_version`: See Version.
- `signature_type`: Cryptosystem ID for the algorithm used to generate the certificate's signature and extensions hash.
- `key_type`: Cryptosystem ID indicating the algorithm and key size for the public key contained in the certificate.
- `role`: Certificate role. Whether the certificate is a root, intermediate, or leaf certificate. The table below lists accepted values.
- `usage`: Certificate usage. Specifies additional operational usage expressed by the certificate, if any. See the table below for defined usage values.
- `lifecycle`: Restricts authentication to a particular PSA lifecycle state. Encoding is the same as the psa_lifecycle data type.
- `oem_constraint`: Customizable constraint bitfield. Semantics are defined by the integrator or OEM and used to apply additional constraints on authentication.
- `extension_bytes`: Size in bytes of the following (non-mandatory) extensions. If there are no extensions, this field must be set to zero.
- `soc_id`: Device unique ID. See the soc_id data type.
- `soc_class`: Vendor-defined device family ID. See the soc_class data type.
- `permissions_mask`: Bit mask limiting the permissions that can be requested in the ADAC Token. A set bit indicates that the permission corresponding to that bit position is allowed by the containing certificate to be requested in the token. The full certificate chain and hardware-defined permissions mask must be taken into account to determine the final permissions mask.

The data layout for the certificate header is shown in the following table.

Table 35 Certificate Header

Word	Byte 0	Byte 1	Byte 2	Byte 3
0	`format_version.major`	`format_version.minor`	`signature_type`	`key_type`
1	`role`	`usage`	`(0)`	`(0)`
2	`lifecycle`		`oem_constraint`	
3	`extension_bytes`			
4	`soc_class`			
5	`soc_id[31:0]`			
6	`soc_id[63:32]`			
7	`soc_id[95:64]`			
8	`soc_id[127:96]`			
9	`permissions_mask[31:0]`			
10	`permissions_mask[63:32]`			
11	`permissions_mask[95:64]`			
12	`permissions_mask[127:96]`			

The following C structures describe the certificate header:

```
struct adac_certificate_header_v1_0 {
    struct adac_version format_version;
    uint8_t signature_type;
    uint8_t key_type;
    uint8_t role;
    uint8_t usage;
    uint8_t _reserved[2];
    uint16_t lifecycle;
    uint16_t oem_constraint;
    uint32_t extensions_bytes;
    uint32_t soc_class;
    uint8_t soc_id[16];
    uint8_t permissions_mask[16];
};

struct adac_certificate_v1_0 {
    struct adac_certificate_header_v1_0 header;
    uint8_t extensions_hash[HASH_LENGTH];
    uint8_t public_key[KEY_LENGTH];
    uint8_t signature[SIGNATURE_LENGTH];
    // array of variable-length adac_tlv structs
    uint8_t extensions_data[ROUND_TO_WORDS(header.extensions_bytes)];
};
```

Currently defined versions of the `adac_certificate_v1_0` structure are as follows.

Table 36 Certificate Version

Version	Description
1.0	Initial version

The `role` member of the header must have one of the following values.

Table 37 Role values

Value	Constant	Description
0x1	`ADAC_ROLE_ROOT`	The certificate is a root certificate
0x2	`ADAC_ROLE_INTERMEDIATE`	The certificate is intermediate
0x3	`ADAC_ROLE_LEAF`	The certificate is a leaf certificate

The `usage` member of the header must have one of the following values.

Table 38 Usage values

Value	Constant	Description
0x0	`ADAC_USAGE_NEUTRAL`	The certificate has no special usage.
0x1	`ADAC_USAGE_STANDARD`	The certificate is only for authentication.
0x2	`ADAC_USAGE_RMA`	The certificate moves the device to the RMA lifecycle state.

2.5.3 Extensions

The extensions for an ADAC Certificate are structured as a TLV sequence. In the `adac_certificate_v1_0` struct, the extensions TLV sequence is placed in the `extensions_data` member.

The size of the extensions data is specified in the certificate header as a number of bytes.

Allowed Extension Types

The following table lists those Type IDs that are accepted in the extensions data for an ADAC Certificate. Any extension value with a Type ID not included within this list will be ignored.

Table 39 Allowed Extension Types

Type ID	Name	Description
0x0005	`target_identity`	Target identity
0x0009	`sw_partition_id`	Software partition ID

All vendor-specific type IDs are allowed.

The sw_partition_id extension specifies a software partition to which access is granted. This extension value can be included more than once, in which case access is granted to all listed software partitions.

2.5.4 Rules

Construction

The hash of the certificate extensions is computed as follows, using the hash algorithm identified by the `header.signature_type` cryptosystem.:

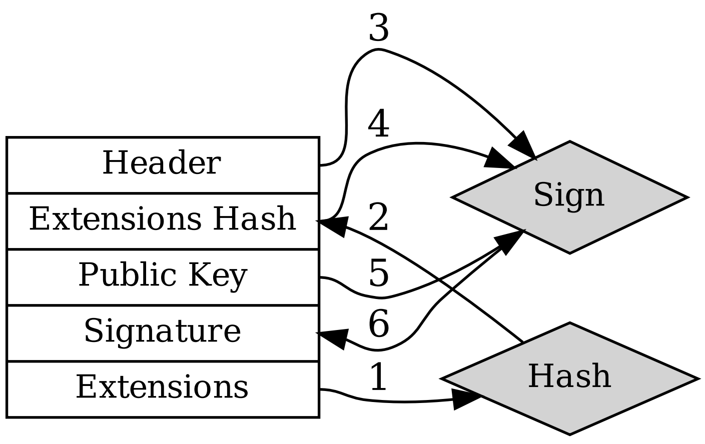
```
extensions_hash = Hash(extensions_data)
```

The signature over the certificate is computed as follows, using the signature algorithm specified by the `header.signature_type` cryptosystem.:

```
signature = Sign(ca.private_key, header || extensions_hash || public_key)
```

In the above expression, the symbol `ca` refers to the signer certificate. For a root certificate, `ca` is the same as the certificate being signed (self-signing).

The following diagram shows the inputs to the certificate signature algorithm.



Validation

The `header.format_version` member can be used to select an appropriate struct definition for parsing the entire header.

These members of the `adac_certificate_v1_0` struct must be validated before any further processing of the certificate is performed:

- `header.format_version`

- `header.signature_type`
- `header.key_type`
- `header.role`
- `header.usage`
- `header.lifecycle`

When processing the complete certificate chain, all non-zero values of a given constraint must be equal in every certificate.

Constraints

When these members of the `adac_certificate_v1_0` struct are set to all zero bits, they are ignored and do not constrain authentication.

- `header.soc_id`
- `header.soc_class`
- `header.lifecycle`
- `header.oem_constraint`

As defined for the psa_lifecycle data type, the `header.lifecycle` is composed of a major PSA state and minor IMPLEMENTATION DEFINED state. If `header.lifecycle` is non-zero, the major state must also be non-zero and authentication is restricted to the specified major state. The minor state is always optional; if non-zero, authentication is restricted to the specified minor state in addition to the major state restriction. If the minor state is zero, then any minor state is allowed. A `header.lifeycle` value with a zero major state and non-zero minor state is invalid.

The `header.oem_constraint` field is compared to the static OEM constraint value provided to the Secure Debug Authenticator by an IMPLEMENTATION DEFINED mechanism. If the two values match then authentication is not constrained, otherwise authentication fails.

2.6 Life-cycle State Command Set

This chapter specifies the life-cycle commands and their parameters. It contains the following sections:

- About the life-cycle state commands
- Status codes
- Life-cycle State Commands
- Change Life-cycle State

2.6.1 About the life-cycle state commands

ADAC defines the following set of commands related to life-cycle states. These commands are defined as a layer building on the Command Protocol.

All commands below are optional in this revision of the specification. However, some commands can return a status code indicating they are unsupported on the target or in the execution environment. This is documented per command.

Detailed specifications for each command follow in this chapter.

Table 40 Life-Cycle state commands

ID	Constant	Command	Description
0x0100	`ADAC_LCS_CHANGE`	Change life-cycle state	Change the life-cycle state of the target

Status codes

The following table lists the complete set of status codes returned by the life-cycle state commands.

Table 41 Life-Cycle change return codes

Status	Constant	Description
0x0000	`ADAC_SUCCESS`	The command has succeeded without error.
0x0001	`ADAC_FAILURE`	The command has failed.
0x0003	`ADAC_UNSUPPORTED`	The command is not supported by the target.
0x0004	`ADAC_UNAUTHORIZED`	The command is not allowed in the current authentication state.
0x0005	`ADAC_INVALID_PARAMETERS`	The command has invalid, inconsistent or missing parameters.
0x7FFF	`ADAC_INVALID_COMMAND`	The command ID is unrecognized.

The status code 0x7FFF is special in that it is not returned by a specific command but instead indicates an unrecognized command ID. See the Error Handling section of the command protocol for more information.

2.6.2 Life-cycle State Commands

Change Life-cycle State

The debug host sends this command to request debug target to change its life-cycle state.

Request

Command ID	`ADAC_LCS_CHANGE` (0x0100)
Request data	TLV sequence

The request data is a TLV sequence and must contain an entry of type psa_lifecycle. The requirements for addtional entries and their type is IMPLEMENTATION DEFINED.

The request sequence should be ordered by increasing ID value. If it is not, some values may be omitted from the results.

Response

Table 43 Life-cycle Transition Response status codes

Status code	Description
`ADAC_SUCCESS`	Life-cycle state transition successful.
`ADAC_FAILURE`	Life-cycle state transition failed.
`ADAC_UNSUPPORTED`	The command is not supported by the target
`ADAC_UNAUTHORIZED`	Life-cycle change is not allowed by current authentication state.
`ADAC_INVALID_PARAMETERS`	Life-cycle state change request is invalid.

The response consists of an optional TLV sequence. If present, it must contain an entry of type psa_lifecycle.

Possible response value type IDs:

The response sequence must be ordered by increasing ID value.

Appendix A: Example System Architectures

This chapter gives several examples of possible system architectures for ADAC. The contains the following sections:

A.1 Example Arm Architecture Externally-hosted Target

This section demonstrates an example mapping of this specification to the Arm architecture. The example shown here focuses on externally hosted debug.

Devices based on the Arm architecture vary considerably depending on the size and complexity of the device. However, there are a number of key components that any device with secure debug will have. These are shown in Example externally-hosted Arm target block diagram..

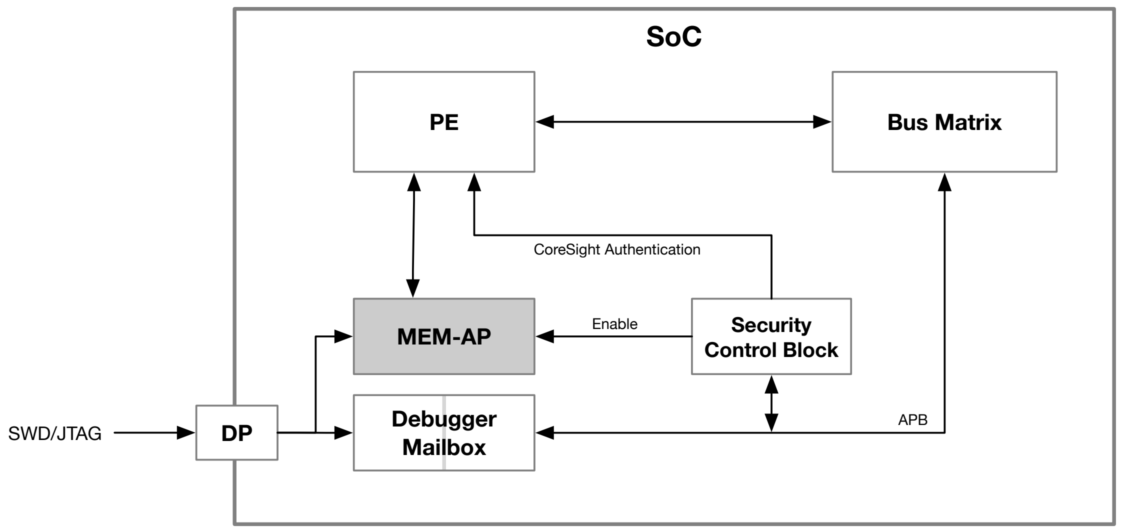


Figure 5 Example externally-hosted Arm target block diagram.

As defined by the Arm® Debug Interface Architecture Specification (ADI), the external interface for debugger access is called the Debug Port (DP). The SWJ-DP is an implementation that supports both SWD and JTAG wire protocols. The DP connects to one or more Access Port (AP) components. A device will have at least one MEM-AP that provides the debugger with the ability to perform transactions on the internal bus fabric and control system and PE-level debug logic.

A standard set of four debug access signals called CoreSight authentication signals are available for controlling the level of debug access for each PE:

- **DBGEN**: Invasive Non-secure debug access
- **NIDEN**: Non-invasive Non-secure debug access
- **SPIDEN**: Invasive Secure debug access, **DBGEN** must also be asserted

- **SPNIDEN**: Non-invasive Secure debug access, **NIDEN** or **DBGEN** must also be asserted

Not all systems will have all four authentication signals. For instance, a system that does not implement a Secure PE will not have **SPIDEN** and **SPNIDEN**.

A debug target in the Secured lifecycle state has these four standard authentication signals disabled by default.

In addition to the standard CoreSight authentication signals, each AP also has a its own enable signal. In a Cortex-M system where the AP is routed through the PE, it is not strictly necessary to disable the AP in the Secured lifecycle state. But in larger systems where a MEM-AP is directly connected to the bus fabric, it is a requirement.

Depending on the debug target's system-level architecture, the debug access signals can be connected in different ways. For instance, a multicore device can expose the CoreSight authentication signals for each PE, or they can be shared. In addition, the debug target can define additional, proprietary security control signals. As an example, a cryptographic accelerator peripheral can accept a signal that controls access to and use of device-unique key(s).

The Security Control Block is IP used by the Secure Debug Authenticator to modify the access control signals. Access to the Security Control Block is restricted to trusted software.

A.1.1 Debugger Mailbox

A type of debugger mailbox that fits well with the ADIv5 architecture is a special type of AP, composed of two sides. One side connects to the DP and is accessible by the debugger as an AP. The other side is a standard APB peripheral. The two sides are themselves connected and can perform byte or word transfers bidirectionally.

A similar debugger mailbox can be built for the ADIv6 architecture, but both sides of the debugger mailbox are APB Completers.

The Arm SDC-600 COM Port is an example of such an debugger mailbox. It is available in versions that support both ADIv5 and ADIv6.

Appendix B: Link Layer

This appendix documents several common link layer protocols used for ADAC communications channels. It contains the following sections:

B.1 About the link layer

The link layer is the protocol layer specific to the communications channel implementation. The primary purpose is to establish a common set of capabilities upon which the higher level protocols can build. It is focused on delivery of packets between the two physical endpoints. Each different communication channel has its own specific link layer protocol.

The link layer provides some or all of these properties for the higher level protocol layers:

- Method to request and initiate communications
- Flow control
- Packetization

Some communication channels have an intrinsic link layer protocol that provides the required properties, and so do not require an additional link layer to be used.

This chapter describes several link layers used for common types of communications channels.

- COM Encapsulation Protocol for Arm SDC-600 COM Port.
- ACK token protocol for common types of proprietary debugger mailbox IP.
- Memory window protocol for devices using a restricted window into system memory.

B.2 Link layers

B.2.1 COM Encapsulation Protocol

The COM Encapsulation Protocol is a byte-oriented protocol for transferring data packets across a COM Port interface such as the Arm SDC-600. The protocol is fully defined in the Arm Advanced Communications Channel Architecture Specification [IHI0076A].

A protocol discovery sequence is defined as part of the COM Encapsulation Protocol. This allows the target to report a unique protocol ID to the host to declare its expected higher level protocol using a method independent of the higher level protocol.

For ADAC, the unique protocol ID is 4 bytes in length, and is defined as shown here:

Format	Protocol ID
ASCII	`ADAC`
Byte sequence	`[0x41 0x44 0x41 0x43]`

> **Note:**
>
> This protocol ID is currently provisional.

Any other protocol ID response sent by the target indicates that a different command protocol is expected, and the host should behave accordingly.

The protocol discovery sequence as byte values are defined as follows. The indicated direction is relative to the host being Requester.

Direction	Protocol	Data
Request	IDR	`[0xA0]`
Response	IDA	`[0xA1]`
	Protocol ID	`[0x41 0x44 0x41 0x43]`
	END	`[0xAD]`

The COM Encapsulation Protocol fully specifies the method of packetizing the higher level command requests and responses transferred by the ADAC Command Protocol.

Command protocol requests and replies must be sent in least-significant byte-first (LSB-first) order.

B.2.2 ACK Token

Some SoCs include a simple debugger mailbox that implements transfer of a single 32-bit word at a time in either direction. Most often the hardware does not mandate a particular link layer protocol. For such IP, the ACK token protocol can be used to provide a software-implemented flow control mechanism. An alternative protocol is to make use of status flags in the IP registers, if provided.

In response to each request or reply data word, the other side must send an ACK Token, used for flow control. The request and reply header words do not require an ACK Token to be sent; the reply acts as the ACK for the request.

The upper 16-bits are set by the receiver (the side sending the ACK Token) with number of remaining words expected to be sent. The lower 16-bits are always set to 0xA5A5. This value should be avoided for a valid command ID or command status value.

Byte 0	Byte 1	Byte 2	Byte 3
0xA5	0xA5	`remain_count[7:0]`	`remain_count[15:8]`

The C structure definition for a ACK Token is:

```
struct ack_token {
    uint16_t token;          /* must be set to 0xA5A5 */
    uint16_t remain_count;   /* remaining word count */
};
```

B.2.3 Memory Window

The memory window link layer is designed to be as simple as possible. Performance is not a key concern. The only requirement is that the memory window support both read and write.

```
enum {
    MW_HOST_DONE = 0x12121212,
    MW_TARGET_DONE = 0xEFEFEFEF,
    MW_PATTERN1 = 0xFF00FF00,
    MW_PATTERN2 = 0x00FF00FF
};

struct memory_window {
    uint32t status[4];
    uint32t message[];
};
```

Handshake

- The host initiates, writing to the `status` array the sequence `[MW_PATTERN1, MW_PATTERN2, MW_PATTERN1, MW_PATTERN2]`.
- The target acknowledges, writing to the `status` array the sequence `[MW_PATTERN1, MW_PATTERN2, MW_PATTERN1, MW_PATTERN2]`.

Message exchange

It is assumed that the Command Protocol and its sequence will be used.

- The host writes a Request packet in the `message` member.
- The host writes the `MW_HOST_DONE` value in the `status[0]` member.
- Reading the value `MW_HOST_DONE` in the `status[0]` member signals to the target that it can read the Request packet from the `message` member.
- Once the Request packet is processed, the target writes Response packet in the `message` member.
- The target writes the `MW_TARGET_DONE` value in the `status[0]` member.
- Reading the value `MW_TARGET_DONE` in the `status[0]` member signals to the host that it can read the Response packet from the `message` member.

Note:

Both Request and Response packets encode their respective lengths.

Appendix C: Cryptographic Support

This appendix documents cryptographic suites currently defined for ADAC. It contains the following sections:

- General concepts
- ECDSA
- RSA
- EdDSA (tentative)
- ShangMi - SM2 (tentative)

C.1 General concepts

C.1.1 Algorithm Agility

Algorithm agility is an important feature of security protocols, which is the reason this specification provides a list of different families of asymmetric key cryptographic algorithms. That list of supported algorithms is extensible.

The specification (and reference implementation) recommends and only specifies solutions using the same algorithm and key size. This due to security-sensitive nature of the operations and the constraints of the target-side implementations of the protocols. Supporting multiple cryptographic algorithms and key sizes adds complexity and code size.

C.1.2 Key Sizes

We have defined for each cryptographic algorithm two public key sizes one that matches the minimum publicly recommended sizes, as well as higher level for high or long term assurance.

C.1.3 Hash Function

With each public key algorithm and key size is associated a hash function.

C.2 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) as defined in [FIPS-186-4] and [X9.62-2005] used in conjunction with the following curves:

- NIST P-256 curve (also designated `secp256r1` in [SECGv2] or `prime256v1` in [X9.62-2005])
- NIST P-521 curve (also designated `secp521r1` in [SECGv2])

The public keys are encoded in uncompressed format (without the `0x04` typically used in formats that allow compressed representations).

The signatures do not include the ASN.1 DER encoding.

We recommend using the deterministic variant of ECDSA (see [RFC6979]) for generating signature, this is transparent to the implementation on the target.

C.2.1 P-256 Curve

```
#define ECDSA_P256_PUBLIC_KEY_SIZE 64
#define ECDSA_P256_SIGNATURE_SIZE  64
#define ECDSA_P256_HASH_SIZE       32
#define ECDSA_P256_HASH_ALGORITHM  PSA_ALG_SHA_256
#define ECDSA_P256_SIGN_ALGORITHM  PSA_ALG_DETERMINISTIC_ECDSA(PSA_ALG_SHA_256)

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[ECDSA_P256_PUBLIC_KEY_SIZE]; // P-256 public key
    uint8_t extensions_hash[ECDSA_P256_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[ECDSA_P256_SIGNATURE_SIZE]; // P-256 with SHA-256 signature
    uint32_t extensions[];
} certificate_p256_p256_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[ECDSA_P256_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[ECDSA_P256_SIGNATURE_SIZE]; // P-256 with SHA-256 signature
    uint32_t extensions[];
} token_p256_t;
```

C.2.2 P-521 Curve

```
#define ECDSA_P521_PUBLIC_KEY_SIZE 132
#define ECDSA_P521_SIGNATURE_SIZE  132
#define ECDSA_P521_HASH_SIZE       64
#define ECDSA_P521_HASH_ALGORITHM  PSA_ALG_SHA_512
#define ECDSA_P521_SIGN_ALGORITHM  PSA_ALG_DETERMINISTIC_ECDSA(PSA_ALG_SHA_512)

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[ROUND_TO_WORD(ECDSA_P521_PUBLIC_KEY_SIZE)]; // P-521 public key
    uint8_t extensions_hash[ECDSA_P521_HASH_SIZE]; // SHA-512 hash
    uint8_t signature[ROUND_TO_WORD(ECDSA_P521_SIGNATURE_SIZE)]; // P-521 with SHA-512 signature
    uint32_t extensions[];
} certificate_p521_p521_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[ECDSA_P521_HASH_SIZE]; // SHA-512 hash
    uint8_t signature[ECDSA_P521_SIGNATURE_SIZE]; // P-521 with SHA-512 signature
```

(continues on next page)

(continued from previous page)

```
    uint32_t extensions[];
} token_p521_t;
```

C.3 RSA

RSA (see [RFC8017]) is included in specification.

This specification forces the the use of the value `F4` (`65537` in decimal, `0x10001` in hexadecimal) for exponent.

The public keys are encoded as the raw value of the modulus (without the leading zero mandated by ASN.1 DER encoding).

Signatures use the Probabilistic Signature Scheme.

C.3.1 RSA 3072-bit keys

```
#define RSA_3072_PUBLIC_KEY_SIZE 384
#define RSA_3072_SIGNATURE_SIZE  384
#define RSA_3072_HASH_SIZE       32
#define RSA_3072_HASH_ALGORITHM  PSA_ALG_SHA_256
#define RSA_3072_SIGN_ALGORITHM  PSA_ALG_RSA_PSS(PSA_ALG_SHA_256)

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[RSA_3072_PUBLIC_KEY_SIZE]; // RSA 3072-bit public key
    uint8_t extensions_hash[RSA_3072_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[RSA_3072_SIGNATURE_SIZE]; // RSA with SHA-256 signature
    uint32_t extensions[];
} certificate_rsa3072_rsa3072_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[RSA_3072_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[RSA_3072_SIGNATURE_SIZE]; // RSA with SHA-256 signature
    uint32_t extensions[];
} token_rsa3072_t;
```

C.3.2 RSA 4096-bit keys

```
#define RSA_4096_PUBLIC_KEY_SIZE 512
#define RSA_4096_SIGNATURE_SIZE  512
#define RSA_4096_HASH_SIZE       32
#define RSA_4096_HASH_ALGORITHM  PSA_ALG_SHA_256
#define RSA_4096_SIGN_ALGORITHM  PSA_ALG_RSA_PKCS1V15_SIGN(PSA_ALG_SHA_256)

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[RSA_4096_PUBLIC_KEY_SIZE]; // RSA 4096-bit public key
    uint8_t extensions_hash[RSA_4096_HASH_SIZE]; // SHA-256 hash
```

(continues on next page)

(continued from previous page)

```
    uint8_t signature[RSA_4096_SIGNATURE_SIZE]; // RSA with SHA-256 signature
    uint32_t extensions[];
} certificate_rsa4096_rsa4096_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[RSA_4096_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[RSA_4096_SIGNATURE_SIZE]; // RSA with SHA-256 signature
    uint32_t extensions[];
} token_rsa4096_t;
```

C.4 EdDSA (tentative)

Edwards-Curve Digital Signature Algorithm (EdDSA), see [RFC8032].

C.4.1 Ed25519 Curve

See [Ed25519].

```
#define EDDSA_ED25519_PUBLIC_KEY_SIZE 32
#define EDDSA_ED25519_SIGNATURE_SIZE  64
#define EDDSA_ED25519_HASH_SIZE       64
#define EDDSA_ED25519_HASH_ALGORITHM  PSA_ALG_SHA_512
#define EDDSA_ED25519_SIGN_ALGORITHM  PSA_ALG_ED25519PH

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[ROUND_TO_WORD(EDDSA_ED25519_PUBLIC_KEY_SIZE)];
    uint8_t extensions_hash[EDDSA_ED25519_HASH_SIZE];
    uint8_t signature[ROUND_TO_WORD(EDDSA_ED25519_SIGNATURE_SIZE)];
    uint32_t extensions[];
} certificate_ed255_ed255_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[EDDSA_ED25519_HASH_SIZE]; // SHA-512 hash
    uint8_t signature[EDDSA_ED25519_SIGNATURE_SIZE]; // Ed25519 signature
    uint32_t extensions[];
} token_ed255_t;
```

C.4.2 Ed448 Curve

See [Ed448].

```
#define EDDSA_ED448_PUBLIC_KEY_SIZE 57
#define EDDSA_ED448_SIGNATURE_SIZE  114
#define EDDSA_ED448_HASH_SIZE       64
#define EDDSA_ED448_HASH_ALGORITHM  PSA_ALG_SHAKE256_512
#define ECDSA_ED448_SIGN_ALGORITHM  PSA_ALG_ED448PH
```

(continues on next page)

(continued from previous page)

```
typedef struct {
    certificate_header_t header;
    uint8_t pubkey[ROUND_TO_WORD(EDDSA_ED448_PUBLIC_KEY_SIZE)];
    uint8_t extensions_hash[EDDSA_ED448_HASH_SIZE];
    uint8_t signature[ROUND_TO_WORD(EDDSA_ED448_SIGNATURE_SIZE)];
    uint32_t extensions[];
} certificate_ed448_ed448_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[EDDSA_ED448_HASH_SIZE]; // SHAKE256 hash
    uint8_t signature[EDDSA_ED448_SIGNATURE_SIZE]; // Ed448 signature
    uint32_t extensions[];
} token_ed448_t;
```

C.5 ShangMi - SM2 (tentative)

SM2 is a set of elliptic curve based cryptographic algorithms including digital signature (see [ISO-SM2]). SM2 is used with the SM3 hash function (see [ISO-SM3]).

C.5.1 SM2

```
#define SM2_SM3_PUBLIC_KEY_SIZE 64
#define SM2_SM3_SIGNATURE_SIZE  64
#define SM2_SM3_HASH_SIZE       32
#define SM2_SM3_HASH_ALGORITHM  PSA_ALG_SM3
#define SM2_SM3_SIGN_ALGORITHM  PSA_ALG_SM2 // Not defined yet

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[SM2_SM3_PUBLIC_KEY_SIZE]; // SM2 public key
    uint8_t extensions_hash[SM2_SM3_HASH_SIZE]; // SM3 hash
    uint8_t signature[SM2_SM3_SIGNATURE_SIZE]; // SM2 with SM3 signature
    uint32_t extensions[];
} certificate_sm2sm3_sm2sm3_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[SM2_SM3_HASH_SIZE]; // SM3 hash
    uint8_t signature[SM2_SM3_SIGNATURE_SIZE]; // SM2 with SM3 signature
    uint32_t extensions[];
} token_sm2sm3_t;
```

C.6 Secret key algorithms (tentative)

C.6.1 CMAC with AES

CMAC with AES is authentication algorithm based on CMAC with the 128-bit Advanced Encryption Standard (AES), see [SP800-38B] and [RFC4493].

```
#define CMAC_PUBLIC_KEY_SIZE 16
#define CMAC_SIGNATURE_SIZE  16
#define CMAC_HASH_SIZE       16
#define CMAC_HASH_ALGORITHM  PSA_ALG_CMAC
#define CMAC_SIGN_ALGORITHM  PSA_ALG_CMAC

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[CMAC_PUBLIC_KEY_SIZE]; // Nonce
    uint8_t extensions_hash[CMAC_HASH_SIZE]; // CMAC
    uint8_t signature[CMAC_SIGNATURE_SIZE]; // CMAC
    uint32_t extensions[];
} certificate_cmac_cmac_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[CMAC_HASH_SIZE]; // CMAC
    uint8_t signature[CMAC_SIGNATURE_SIZE]; // CMAC
    uint32_t extensions[];
} token_cmac_t;
```

C.6.2 HMAC with SHA-256

HMAC a mechanism for message authentication using cryptographic hash functions. See [RFC2104], [RFC6234] and [SP800-107r1].

```
#define HMAC_PUBLIC_KEY_SIZE 32
#define HMAC_SIGNATURE_SIZE  32
#define HMAC_HASH_SIZE       32
#define HMAC_HASH_ALGORITHM  PSA_ALG_SHA_256
#define HMAC_SIGN_ALGORITHM  PSA_ALG_HMAC(PSA_ALG_SHA_256)

typedef struct {
    certificate_header_t header;
    uint8_t pubkey[HMAC_PUBLIC_KEY_SIZE]; // Nonce
    uint8_t extensions_hash[HMAC_HASH_SIZE]; // SHA-256 hash
    uint8_t signature[HMAC_SIGNATURE_SIZE]; // HMAC-SHA-256
    uint32_t extensions[];
} certificate_hmac_hmac_t;

typedef struct {
    token_header_t header;
    uint8_t extensions_hash[HMAC_HASH_SIZE]; // SHA-256 Hash
    uint8_t signature[HMAC_SIGNATURE_SIZE]; // HMAC-SHA-256
    uint32_t extensions[];
} token_hmac_t;
```

Appendix D: Security Risk Assessment

This appendix contains a simplified Security Risk Assessment for ADAC.

See chapter Security Model for more details about:

- Use Cases
- Stakeholders
- Security Goals
- Assets and Actors
- Trust Boundaries
- Assumptions

D.1 Threat Model

D.1.1 Attack Surface and Adversarial Model

The attack surface is limited to the link between the two sides of the protocol. Attackers are assumed to have obtained access to that link, allowing them to read, modify, spoof, replay messages, or issue malformed commands.

Out of scope: debug probes attached to connected devices that export the debug link layer.

D.1.2 Threats and Attacks

Attacks on protocols can insert, delete, swap, modify, or replay messages. The aim is to force an illegal state change, swap bits of the debug vector, or disable the debug protection engine.

Another class of attacks is denial of service. Those attacks could exhaust resources on the target or disable enough functionalities to render it inoperable.

Message-based attacks

- Malformed `_reserved` fields should have no impact on exchanges since they are always fixed-sized and not meant to be parsed.
- `status` fields can be modified to trigger inconsistencies in protocol exchanges, possibly leading to deadlocks.
- `data_count` can be modified to erroneous values, leading a peer to either wait for more incoming data, or to truncate the message.
- TLV fields should be analyzed with caution as their maximum size is defined on 32 bits, possibly leading to peers expecting gigabytes of data in messages.

ADAC messages are built as a status + payload structure containing potentially free-form TLV fields. Implementers should be cautious in parsing any data received from the host side.

Attacks on Credentials

ADAC credentials are signed, i.e. modifying any single bit should render the signature invalid and prevent any successful authentication. Attacks on signature schemes can take two forms:

- Finding the private key associated to a known public key
- Finding collisions in the underlying hash function

ADAC is restricted to signature and hashing algorithms which have, at the time of writing, no such known flaws. See chapter Key and Signature Types for more details. As long as the chosen algorithms are resistant to the above-mentioned attacks, it should not be possible to forge a signature or use an existing signature for another set of credentials.

The ADAC protocol requires credentials to be presented by the host starting from the root, up to possible intermediate, and end with the leaf certificate. An attacker could abuse that convention to submit a seemingly infinite list of certificates to the target, possibly leading to resource exhaustion.

D.1.3 Risks and Mitigations

Denial of Service

Modifying messages or inserting garbage into an exchange is only likely to lead to an aborted authentication. In the worst case, peers can both be waiting for their counterpart to send more data, leading to a deadlock.

Mitigation

Targets need to be defensive against any kind of data received from the host, aborting communication early whenever issues are met. The state machine does not include any error state. Failure should always lead back to the target listening for incoming requests.

Resource consumption

Forcing the target to generate a very large number of challenges may deplete its entropy source.

Mitigation

This can be thwarted with challenges based on a properly seeded DRBG, provided the target is not reset before saving the seed. Another way would be to rate-limit authentication attempts on the target side. For example: after three failed authentication attempts the target waits one minute before it agrees to enter a new authentication exchange, forcing an attacker to either wait or reset the device.

Triggering failures

Using malformed credentials to trigger errors in credential parsing on the target side, possibly leading to wrong settings in the debug vector.

Mitigation

Credential parsing needs to be extremely strict about data sizes and bail out early in case of errors. Setting the final debug vector should only happen after all credentials have been verified.

Weak cryptography

Cryptographic algorithms invariably get broken over time: new types of attacks, more powerful machines used for brute force.

Mitigation

Adding support for multiple cryptosystems for authentication enables easy switching when necessary. Beyond that, firmware updates must be able to take care of replacing broken code and updating root authority credentials.

Appendix E: Changes in this document

Changes between `1.0.0` and `1.0.1`:

- Replaced mentions of 32-bit word sizes with bytes

Changes from the previous `1.0-beta2` version:

- Changed documentation format
- Updated Reference table
- Moved and expanded terms and abbreviations
- Switched to inclusive terms
- Renamed namespace prefixes from `SDP_` to `ADAC_`
- Added example for Discovery
- Added `ADAC_USAGE_NEUTRAL` to certificate usage
- Moved Security Risk Assessment to its own section in Appendix