Stream: Internet Engineering Task Force (IETF)
RFC: 9200
Category: Standards Track
Published: August 2022
ISSN: 2070-1721
Authors: L. Seitz *Combitech*; G. Selander *Ericsson*; E. Wahlstroem; S. Erdtman *Spotify AB*; H. Tschofenig *Arm Ltd.*

RFC 9200
Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth)

Abstract

This specification defines a framework for authentication and authorization in Internet of Things (IoT) environments called ACE-OAuth. The framework is based on a set of building blocks including OAuth 2.0 and the Constrained Application Protocol (CoAP), thus transforming a well-known and widely used authorization solution into a form suitable for IoT devices. Existing specifications are used where possible, but extensions are added and profiles are defined to better serve the IoT use cases.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9200.

Copyright Notice

Table of Contents

1. Introduction

Authorization is the process for granting approval to an entity to access a generic resource [RFC4949]. The authorization task itself can best be described as granting access to a requesting client for a resource hosted on a device, i.e., the resource server (RS). This exchange is mediated by one or multiple authorization servers (ASes). Managing authorization for a large number of devices and users can be a complex task.

While prior work on authorization solutions for the Web and for the mobile environment also applies to the Internet of Things (IoT) environment, many IoT devices are constrained, for example, in terms of processing capabilities, available memory, etc. For such devices, the Constrained Application Protocol (CoAP) [RFC7252] can alleviate some resource concerns when used instead of HTTP to implement the communication flows of this specification.

Appendix A gives an overview of the constraints considered in this design, and a more detailed treatment of constraints can be found in [RFC7228]. This design aims to accommodate different IoT deployments as well as a continuous range of device and network capabilities. Taking energy consumption as an example, at one end, there are energy-harvesting or battery-powered devices that have a tight power budget; on the other end, there are mains-powered devices; and all levels exist in between.

Hence, IoT devices may be very different in terms of available processing and message exchange capabilities, and there is a need to support many different authorization use cases [RFC7744].

This specification describes a framework for Authentication and Authorization for Constrained Environments (ACE) built on reuse of OAuth 2.0 [RFC6749], thereby extending authorization to Internet of Things devices. This specification contains the necessary building blocks for adjusting OAuth 2.0 to IoT environments.

Profiles of this framework are available in separate specifications, such as [RFC9202] or [RFC9203]. Such profiles may specify the use of the framework for a specific security protocol and the underlying transports for use in a specific deployment environment to improve interoperability. Implementations may claim conformance with a specific profile, whereby implementations utilizing the same profile interoperate, while implementations of different profiles are not expected to be interoperable. More powerful devices, such as mobile phones and tablets, may implement multiple profiles and will therefore be able to interact with a wider range of constrained devices. Requirements on profiles are described at contextually appropriate places throughout this specification and also summarized in Appendix C.

2. Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Certain security-related terms, such as "authentication", "authorization", "confidentiality", "(data) integrity", "message authentication code", and "verify", are taken from [RFC4949].

Since exchanges in this specification are described as RESTful protocol interactions, HTTP [RFC9110] offers useful terminology. (Note that "RESTful" refers to the Representational State Transfer (REST) architecture.)

Terminology for entities in the architecture is defined in OAuth 2.0 [RFC6749], such as client (C), resource server (RS), and authorization server (AS).

Note that the term "endpoint" is used here following its OAuth definition, which is to denote resources, such as token and introspection at the AS and authz-info at the RS (see Section 5.10.1 for a definition of the authz-info endpoint). The CoAP definition, which is "[a]n entity participating in the CoAP protocol" [RFC7252], is not used in this specification.

The specification in this document is called the "framework" or "ACE framework". When referring to "profiles of this framework", it refers to additional specifications that define the use of this specification with concrete transport and communication security protocols (e.g., CoAP over DTLS).

The term "Access Information" is used for parameters, other than the access token, provided to the client by the AS to enable it to access the RS (e.g., public key of the RS or profile supported by RS).

The term "authorization information" is used to denote all information, including the claims of relevant access tokens, that an RS uses to determine whether an access request should be granted.

Throughout this document, examples for CBOR data items are expressed in CBOR extended diagnostic notation as defined in Section 8 of [RFC8949] and Appendix G of [RFC8610] ("diagnostic notation"), unless noted otherwise. We often use diagnostic notation comments to provide a textual representation of the numeric parameter names and values.

3. Overview

This specification defines the ACE framework for authorization in the Internet of Things environment. It consists of a set of building blocks.

The basic block is the OAuth 2.0 [RFC6749] framework, which enjoys widespread deployment. Many IoT devices can support OAuth 2.0 without any additional extensions, but for certain constrained settings, additional profiling is needed.

Another building block is the lightweight web transfer protocol CoAP [RFC7252], for those communication environments where HTTP is not appropriate. CoAP typically runs on top of UDP, which further reduces overhead and message exchanges. While this specification defines extensions for the use of OAuth over CoAP, other underlying protocols are not prohibited from being supported in the future, such as HTTP/2 [RFC9113], Message Queuing Telemetry Transport (MQTT) [MQTT5.0], Bluetooth Low Energy (BLE) [BLE], and QUIC [RFC9000]. Note that this

document specifies protocol exchanges in terms of RESTful verbs, such as GET and POST. Future profiles using protocols that do not support these verbs **MUST** specify how the corresponding protocol messages are transmitted instead.

A third building block is the Concise Binary Object Representation (CBOR) [RFC8949], for encodings where JSON [RFC8259] is not sufficiently compact. CBOR is a binary encoding designed for small code and message size. Self-contained tokens and protocol message payloads are encoded in CBOR when CoAP is used. When CoAP is not used, the use of CBOR remains **RECOMMENDED**.

A fourth building block is CBOR Object Signing and Encryption (COSE) [RFC8152], which enables object-level layer security as an alternative or complement to transport layer security (DTLS [RFC6347] [RFC9147] or TLS [RFC8446]). COSE is used to secure self-contained tokens, such as proof-of-possession (PoP) tokens, which are an extension to the OAuth bearer tokens. The default token format is defined in CBOR Web Token (CWT) [RFC8392]. Application-layer security for CoAP using COSE can be provided with Object Security for Constrained RESTful Environments (OSCORE) [RFC8613].

With the building blocks listed above, solutions satisfying various IoT device and network constraints are possible. A list of constraints is described in detail in [RFC7228], and a description of how the building blocks mentioned above relate to the various constraints can be found in Appendix A.

Luckily, not every IoT device suffers from all constraints. Nevertheless, the ACE framework takes all these aspects into account and allows several different deployment variants to coexist, rather than mandating a one-size-fits-all solution. It is important to cover the wide range of possible interworking use cases and the different requirements from a security point of view. Once IoT deployments mature, popular deployment variants will be documented in the form of ACE profiles.

3.1. OAuth 2.0

The OAuth 2.0 authorization framework enables a client to obtain scoped access to a resource with the permission of a resource owner. Authorization information, or references to it, is passed between the nodes using access tokens. These access tokens are issued to clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

A number of OAuth 2.0 terms are used within this specification:

Access Tokens:
: Access tokens are credentials needed to access protected resources. An access token is a data structure representing authorization permissions issued by the AS to the client. Access tokens are generated by the AS and consumed by the RS. The access token content is opaque to the client.

Access tokens can have different formats and various methods of utilization (e.g., cryptographic properties) based on the security requirements of the given deployment.

Introspection:
: Introspection is a method for a resource server, or potentially a client, to query the authorization server for the active state and content of a received access token. This is particularly useful in those cases where the authorization decisions are very dynamic and/or where the received access token itself is an opaque reference, rather than a self-contained token. More information about introspection in OAuth 2.0 can be found in [RFC7662].

Refresh Tokens:
: Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token expires or to obtain additional access tokens with identical or narrower scope (such access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (B) in Figure 1).

  A refresh token in OAuth 2.0 is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers. In this framework, refresh tokens are encoded in binary instead of strings, if used.

Proof-of-Possession Tokens:
: A token may be bound to a cryptographic key, which is then used to bind the token to a request authorized by the token. Such tokens are called proof-of-possession tokens (or PoP tokens).

  The proof-of-possession security concept used here assumes that the AS acts as a trusted third party that binds keys to tokens. In the case of access tokens, these so-called PoP keys are then used by the client to demonstrate the possession of the secret to the RS when accessing the resource. The RS, when receiving an access token, needs to verify that the key used by the client matches the one bound to the access token. When this specification uses the term "access token", it is assumed to be a PoP access token unless specifically stated otherwise.

  The key bound to the token (the PoP key) may use either symmetric or asymmetric cryptography. The appropriate choice of the kind of cryptography depends on the constraints of the IoT devices as well as on the security requirements of the use case.

  Symmetric PoP key:
  : The AS generates a random, symmetric PoP key. The key is either stored to be returned on introspection calls or included in the token. Either the whole token or only the key **MUST** be encrypted in the latter case. The PoP key is also returned to client together with the token, protected by the secure channel.

Asymmetric PoP key:
: An asymmetric key pair is generated by the client and the public key is sent to the AS (if it does not already have knowledge of the client's public key). Information about the public key, which is the PoP key in this case, is either stored to be returned on introspection calls or included inside the token and sent back to the client. The resource server consuming the token can identify the public key from the information in the token, which allows the client to use the corresponding private key for the proof of possession.

The token is either a simple reference or a structured information object (e.g., CWT [RFC8392]) protected by a cryptographic wrapper (e.g., COSE [RFC8152]). The choice of PoP key does not necessarily imply a specific credential type for the integrity protection of the token.

Scopes and Permissions:
: In OAuth 2.0, the client specifies the type of permissions it is seeking to obtain (via the `scope` parameter) in the access token request. In turn, the AS may use the `scope` response parameter to inform the client of the scope of the access token issued. As the client could be a constrained device as well, this specification defines the use of CBOR encoding (see Section 5) for such requests and responses.

  The values of the `scope` parameter in OAuth 2.0 are expressed as a list of space-delimited, case-sensitive strings with a semantic that is well known to the AS and the RS. More details about the concept of scopes are found under Section 3.3 of [RFC6749].

Claims:
: Information carried in the access token or returned from introspection, called claims, is in the form of name-value pairs. An access token may, for example, include a claim identifying the AS that issued the token (via the `iss` claim) and what audience the access token is intended for (via the `aud` claim). The audience of an access token can be a specific resource, one resource, or many resource servers. The resource owner policies influence what claims are put into the access token by the authorization server.

  While the structure and encoding of the access token varies throughout deployments, a standardized format has been defined with the JSON Web Token (JWT) [RFC7519], where claims are encoded as a JSON object. In [RFC8392], the CBOR Web Token (CWT) has been defined as an equivalent format using CBOR encoding.

Token and Introspection Endpoints:
: The AS hosts the token endpoint that allows a client to request access tokens. The client makes a POST request to the token endpoint on the AS and receives the access token in the response (if the request was successful).

  In some deployments, a token introspection endpoint is provided by the AS, which can be used by the RS and potentially the client, if they need to request additional information regarding a received access token. The requesting entity makes a POST request to the introspection endpoint on the AS and receives information about the access token in the response. (See "Introspection" above.)

3.2. CoAP

CoAP is an application-layer protocol similar to HTTP but specifically designed for constrained environments. CoAP typically uses datagram-oriented transport, such as UDP, where reordering and loss of packets can occur. A security solution needs to take the latter aspects into account.

While HTTP uses headers and query strings to convey additional information about a request, CoAP encodes such information into header parameters called 'options'.

CoAP supports application-layer fragmentation of the CoAP payloads through block-wise transfers [RFC7959]. However, block-wise transfer does not increase the size limits of CoAP options; therefore, data encoded in options has to be kept small.

Transport layer security for CoAP can be provided by DTLS or TLS [RFC6347] [RFC8446] [RFC9147]. CoAP defines a number of proxy operations that require transport layer security to be terminated at the proxy. One approach for protecting CoAP communication end-to-end through proxies, and also to support security for CoAP over a different transport in a uniform way, is to provide security at the application layer using an object-based security mechanism, such as COSE [RFC8152].

One application of COSE is OSCORE [RFC8613], which provides end-to-end confidentiality, integrity and replay protection, and a secure binding between CoAP request and response messages. In OSCORE, the CoAP messages are wrapped in COSE objects and sent using CoAP.

In this framework, the use of CoAP as replacement for HTTP is **RECOMMENDED** for use in constrained environments. For communication security, this framework does not make an explicit protocol recommendation, since the choice depends on the requirements of the specific application. DTLS [RFC6347] [RFC9147] and OSCORE [RFC8613] are mentioned as examples; other protocols fulfilling the requirements from Section 6.5 are also applicable.

4. Protocol Interactions

The ACE framework is based on the OAuth 2.0 protocol interactions using the token endpoint and optionally the introspection endpoint. A client obtains an access token, and optionally a refresh token, from an AS using the token endpoint and subsequently presents the access token to an RS to gain access to a protected resource. In most deployments, the RS can process the access token locally; however, in some cases, the RS may present it to the AS via the introspection endpoint to get fresh information. These interactions are shown in Figure 1. An overview of various OAuth concepts is provided in Section 3.1.

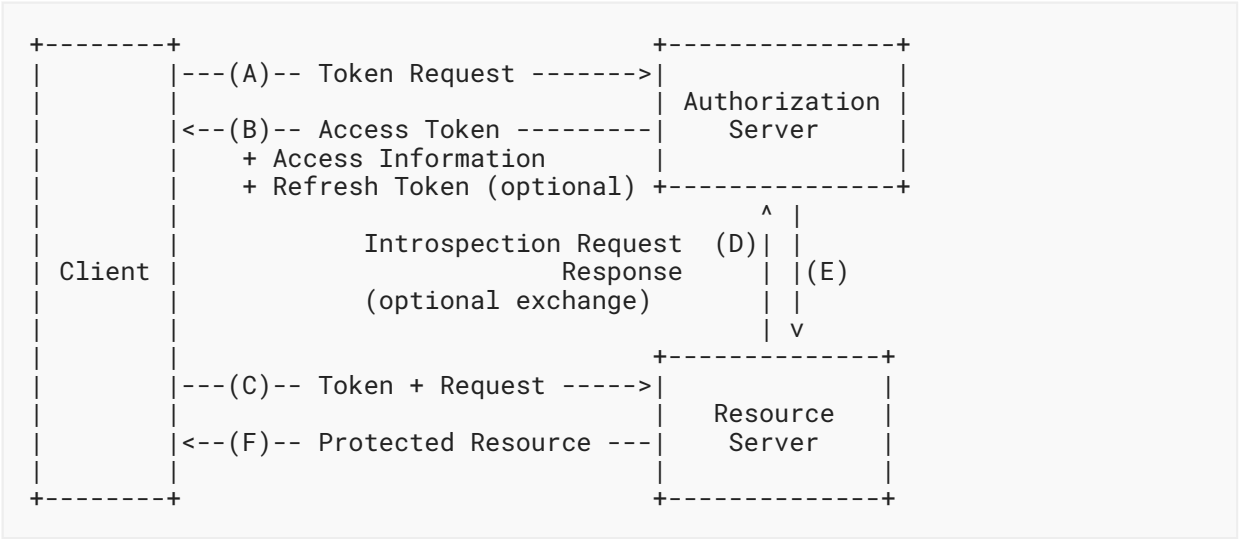

Figure 1: Basic Protocol Flow

Requesting an Access Token (A):
: The client makes an access token request to the token endpoint at the AS. This framework assumes the use of PoP access tokens (see Section 3.1 for a short description) wherein the AS binds a key to an access token. The client may include permissions it seeks to obtain and information about the credentials it wants to use for proof of possession (e.g., symmetric/asymmetric cryptography or a reference to a specific key) of the access token.

Access Token Response (B):
: If the request from the client has been successfully verified, authenticated, and authorized, the AS returns an access token and optionally a refresh token. Note that only certain grant types support refresh tokens. The AS can also return additional parameters, referred to as "Access Information". In addition to the response parameters defined by OAuth 2.0 and the PoP access token extension, this framework defines parameters that can be used to inform the client about capabilities of the RS, e.g., the profile the RS supports. More information about these parameters can be found in Section 5.8.4.

Resource Request (C):
: The client interacts with the RS to request access to the protected resource and provides the access token. The protocol to use between the client and the RS is not restricted to CoAP. HTTP, HTTP/2 [RFC9113], QUIC [RFC9000], MQTT [MQTT5.0], Bluetooth Low Energy [BLE], etc., are also viable candidates.

  Depending on the device limitations and the selected protocol, this exchange may be split up into two parts:

  (1) the client sends the access token containing, or referencing, the authorization information to the RS that will be used for subsequent resource requests by the client, and

(2) the client makes the resource access request using the communication security protocol and other Access Information obtained from the AS.

The client and the RS mutually authenticate using the security protocol specified in the profile (see step (B)) and the keys obtained in the access token or the Access Information. The RS verifies that the token is integrity protected and originated by the AS. It then compares the claims contained in the access token with the resource request. If the RS is online, validation can be handed over to the AS using token introspection (see messages (D) and (E)) over HTTP or CoAP.

Token Introspection Request (D):
: A resource server may be configured to introspect the access token by including it in a request to the introspection endpoint at that AS. Token introspection over CoAP is defined in Section 5.9 and for HTTP in [RFC7662].

  Note that token introspection is an optional step and can be omitted if the token is self-contained and the resource server is prepared to perform the token validation on its own.

Token Introspection Response (E):
: The AS validates the token and returns the most recent parameters, such as `scope`, `audience`, validity, etc., associated with it back to the RS. The RS then uses the received parameters to process the request to either accept or to deny it.

Protected Resource (F):
: If the request from the client is authorized, the RS fulfills the request and returns a response with the appropriate response code. The RS uses the dynamically established keys to protect the response according to the communication security protocol used.

The OAuth 2.0 framework defines a number of "protocol flows" via grant types, which have been extended further with extensions to OAuth 2.0 (such as [RFC7521] and [RFC8628]). What grant type works best depends on the usage scenario; [RFC7744] describes many different IoT use cases, but there are two grant types that cover a majority of these scenarios, namely the authorization code grant (described in Section 4.1 of [RFC6749]) and the client credentials grant (described in Section 4.4 of [RFC6749]). The authorization code grant is a good fit for use with apps running on smartphones and tablets that request access to IoT devices, a common scenario in the smart home environment, where users need to go through an authentication and authorization phase (at least during the initial setup phase). The native apps guidelines described in [RFC8252] are applicable to this use case. The client credentials grant is a good fit for use with IoT devices where the OAuth client itself is constrained. In such a case, the resource owner has prearranged access rights for the client with the authorization server, which is often accomplished using a commissioning tool.

The consent of the resource owner, for giving a client access to a protected resource, can be provided dynamically as in the classical OAuth flows, or it could be preconfigured by the resource owner as authorization policies at the AS, which the AS evaluates when a token request arrives. The resource owner and the requesting party (i.e., client owner) are not shown in Figure 1.

This framework supports a wide variety of communication security mechanisms between the ACE entities, such as the client, AS, and RS. It is assumed that the client has been registered (also called enrolled or onboarded) to an AS using a mechanism defined outside the scope of this document. In practice, various techniques for onboarding have been used, such as factory-based provisioning or the use of commissioning tools. Regardless of the onboarding technique, this provisioning procedure implies that the client and the AS exchange credentials and configuration parameters. These credentials are used to mutually authenticate each other and to protect messages exchanged between the client and the AS.

It is also assumed that the RS has been registered with the AS, potentially in a similar way as the client has been registered with the AS. Established keying material between the AS and the RS allows the AS to apply cryptographic protection to the access token to ensure that its content cannot be modified and, if needed, that the content is confidentiality protected. Confidentiality protection of the access token content would be provided on top of confidentiality protection via a communication security protocol.

The keying material necessary for establishing communication security between the C and RS is dynamically established as part of the protocol described in this document.

At the start of the protocol, there is an optional discovery step where the client discovers the resource server and the resources this server hosts. In this step, the client might also determine what permissions are needed to access the protected resource. A generic procedure is described in Section 5.1; profiles **MAY** define other procedures for discovery.

In Bluetooth Low Energy, for example, advertisements are broadcast by a peripheral, including information about the primary services. In CoAP, as a second example, a client can make a request to "/.well-known/core" to obtain information about available resources, which are returned in a standardized format, as described in [RFC6690].

5. Framework

The following sections detail the profiling and extensions of OAuth 2.0 for constrained environments, which constitutes the ACE framework.

Credential Provisioning
: In constrained environments, it cannot be assumed that the client and the RS are part of a common key infrastructure. Therefore, the AS provisions credentials and associated information to allow mutual authentication between the client and the RS. The resulting security association between the client and the RS may then also be used to bind these credentials to the access tokens the client uses.

Proof of Possession
: The ACE framework, by default, implements proof of possession for access tokens, i.e., that the token holder can prove being a holder of the key bound to the token. The binding is provided by the `cnf` (confirmation) claim [RFC8747], indicating what key is used for proof of possession. If a client needs to submit a new access token, e.g., to obtain additional access rights, they can request that the AS binds this token to the same key as the previous one.

ACE Profiles
: The client or RS may be limited in the encodings or protocols it supports. To support a variety of different deployment settings, specific interactions between the client and RS are defined in an ACE profile. In the ACE framework, the AS is expected to manage the matching of compatible profile choices between a client and an RS. The AS informs the client of the selected profile using the `ace_profile` parameter in the token response.

OAuth 2.0 requires the use of TLS to protect the communication between the AS and client when requesting an access token between the client and RS when accessing a resource and between the AS and RS if introspection is used. In constrained settings, TLS is not always feasible or desirable. Nevertheless, it is **REQUIRED** that the communications named above are encrypted, integrity protected, and protected against message replay. It is also **REQUIRED** that the communicating endpoints perform mutual authentication. Furthermore, it **MUST** be assured that responses are bound to the requests in the sense that the receiver of a response can be certain that the response actually belongs to a certain request. Note that setting up such a secure communication may require some unprotected messages to be exchanged first (e.g., sending the token from the client to the RS).

Profiles **MUST** specify a communication security protocol between the client and RS that provides the features required above. Profiles **MUST** specify a communication security protocol **RECOMMENDED** to be used between the client and AS that provides the features required above. Profiles **MUST** specify, for introspection, a communication security protocol **RECOMMENDED** to be used between the RS and AS that provides the features required above. These recommendations enable interoperability between different implementations without the need to define a new profile if the communication between the C and AS, or between the RS and AS, is protected with a different security protocol complying with the security requirements above.

In OAuth 2.0, the communication with the Token and the Introspection endpoints at the AS is assumed to be via HTTP and may use Uri-query parameters. When profiles of this framework use CoAP instead, it is **REQUIRED** to use of the following alternative instead of Uri-query parameters: The sender (client or RS) encodes the parameters of its request as a CBOR map and submits that map as the payload of the POST request. The CBOR encoding for a number of OAuth 2.0 parameters is specified in this document; if a profile needs to use other OAuth 2.0 parameters with CoAP, it **MUST** specify their CBOR encoding.

Profiles that use CBOR encoding of protocol message parameters at the outermost encoding layer **MUST** use the Content-Format "application/ace+cbor". If CoAP is used for communication, the Content-Format **MUST** be abbreviated with the ID: 19 (see Section 8.16).

The OAuth 2.0 AS uses a JSON structure in the payload of its responses both to the client and RS. If CoAP is used, it is **REQUIRED** to use CBOR [RFC8949] instead of JSON. Depending on the profile, the CBOR payload **MAY** be enclosed in a non-CBOR cryptographic wrapper.

5.1. Discovering Authorization Servers

The C must discover the AS in charge of the RS to determine where to request the access token. To do so, the C 1) must find out the AS URI to which the token request message must be sent and 2) **MUST** validate that the AS with this URI is authorized to provide access tokens for this RS.

In order to determine the AS URI, the C **MAY** send an initial Unauthorized Resource Request message to the RS. The RS then denies the request and sends the address of its AS back to the C (see Section 5.2). How the C validates the AS authorization is not in scope for this document. The C may, for example, ask its owner if this AS is authorized for this RS. The C may also use a mechanism that addresses both problems at once (e.g., by querying a dedicated secure service provided by the client owner) .

5.2. Unauthorized Resource Request Message

An Unauthorized Resource Request message is a request for any resource hosted by the RS for which the client does not have authorization granted. The RSs **MUST** treat any request for a protected resource as an Unauthorized Resource Request message when any of the following hold:

- The request has been received on an unsecured channel.
- The RS has no valid access token for the sender of the request regarding the requested action on that resource.
- The RS has a valid access token for the sender of the request, but that token does not authorize the requested action on the requested resource.

Note: These conditions ensure that the RS can handle requests autonomously once access was granted and a secure channel has been established between the C and RS. The authz-info endpoint, as part of the process for authorizing to protected resources, is not itself a protected resource and **MUST NOT** be protected as specified above (cf. Section 5.10.1).

Unauthorized Resource Request messages **MUST** be denied with an "unauthorized_client" error response. In this response, the resource server **SHOULD** provide proper AS Request Creation Hints to enable the client to request an access token from the RS's AS, as described in Section 5.3.

The handling of all client requests (including unauthorized ones) by the RS is described in Section 5.10.2.

5.3. AS Request Creation Hints

The AS Request Creation Hints are sent by an RS as a response to an Unauthorized Resource Request message (see Section 5.2) to help the sender of the Unauthorized Resource Request message acquire a valid access token. The AS Request Creation Hints are a CBOR or JSON map, with an **OPTIONAL** element `AS` specifying an absolute URI (see Section 4.3 of [RFC3986]) that identifies the appropriate AS for the RS.

The message can also contain the following **OPTIONAL** parameters:

- An `audience` element contains an identifier the client should request at the AS, as suggested by the RS. With this parameter, when included in the access token request to the AS, the AS is able to restrict the use of the access token to specific RSs. See Section 6.9 for a discussion of this parameter.
- A `kid` (key identifier) element contains the key identifier of a key used in an existing security association between the client and the RS. The RS expects the client to request an access token bound to this key in order to avoid having to reestablish the security association.
- A `cnonce` element contains a client-nonce. See Section 5.3.1.
- A `scope` element contains the suggested scope that the client should request towards the AS.

Table 1 summarizes the parameters that may be part of the AS Request Creation Hints.

Name	CBOR Key	Value Type
AS	1	text string
kid	2	byte string
audience	5	text string
scope	9	text or byte string
cnonce	39	byte string

Table 1: AS Request Creation Hints

Note that the schema part of the AS parameter may need to be adapted to the security protocol that is used between the client and the AS. Thus, the example AS value "coap://as.example.com/token" might need to be transformed to "coaps://as.example.com/token". It is assumed that the client can determine the correct schema part on its own depending on the way it communicates with the AS.

Figure 2 shows an example for an AS Request Creation Hints payload using diagnostic notation.

```
4.01 Unauthorized
Content-Format: application/ace+cbor
Payload :
{
 / AS / 1 : "coaps://as.example.com/token",
 / audience / 5 : "coaps://rs.example.com",
 / scope / 9 : "rTempC",
 / cnonce / 39 : h'e0a156bb3f'
}
```

Figure 2: AS Request Creation Hints Payload Example

In the example above, the response parameter `AS` points the receiver of this message to the URI "coaps://as.example.com/token" to request access tokens. The RS sending this response uses an internal clock that is not synchronized with the clock of the AS. Therefore, it cannot reliably verify the expiration time of access tokens it receives. Nevertheless, to ensure a certain level of access token freshness, the RS has included a `cnonce` parameter (see Section 5.3.1) in the response. (The hex sequence of the `cnonce` parameter is encoded in CBOR-based notation in this example.)

Figure 3 illustrates the mandatory use of binary encoding of the message payload shown in Figure 2.

```
a4                                  # map(4)
   01                               # unsigned(1) (=AS)
   78 1c                            # text(28)
      636f6170733a2f2f61732e657861
      6d706c652e636f6d2f746f6b656e  # "coaps://as.example.com/token"
   05                               # unsigned(5) (=audience)
   76                               # text(22)
      636f6170733a2f2f72732e657861
      6d706c652e636f6d              # "coaps://rs.example.com"
   09                               # unsigned(9) (=scope)
   66                               # text(6)
      7254656d7043                  # "rTempC"
   18 27                            # unsigned(39) (=cnonce)
   45                               # bytes(5)
      e0a156bb3f                    #
```

Figure 3: AS Request Creation Hints Example Encoded in CBOR

5.3.1. The Client-Nonce Parameter

If the RS does not synchronize its clock with the AS, it could be tricked into accepting old access tokens that are either expired or have been compromised. In order to ensure some level of token freshness in that case, the RS can use the `cnonce` (client-nonce) parameter. The processing requirements for this parameter are as follows:

- An RS sending a `cnonce` parameter in an AS Request Creation Hints message **MUST** store information to validate that a given cnonce is fresh. How this is implemented internally is

out of scope for this specification. Expiration of client-nonces should be based roughly on the time it would take a client to obtain an access token after receiving the AS Request Creation Hints, with some allowance for unexpected delays.

- A client receiving a cnonce parameter in an AS Request Creation Hints message **MUST** include this in the parameters when requesting an access token at the AS, using the cnonce parameter from Section 5.8.4.4.
- If an AS grants an access token request containing a cnonce parameter, it **MUST** include this value in the access token, using the cnonce claim specified in Section 5.10.
- An RS that is using the client-nonce mechanism and that receives an access token **MUST** verify that this token contains a cnonce claim, with a client-nonce value that is fresh according to the information stored at the first step above. If the cnonce claim is not present or if the cnonce claim value is not fresh, the RS **MUST** discard the access token. If this was an interaction with the authz-info endpoint, the RS **MUST** also respond with an error message using a response code equivalent to the CoAP code 4.01 (Unauthorized).

5.4. Authorization Grants

To request an access token, the client obtains authorization from the resource owner or uses its client credentials as a grant. The authorization is expressed in the form of an authorization grant.

The OAuth framework [RFC6749] defines four grant types. The grant types can be split up into two groups: those granted on behalf of the resource owner (password, authorization code, implicit) and those for the client (client credentials). Further grant types have been added later, such as an assertion-based authorization grant defined in [RFC7521].

The grant type is selected depending on the use case. In cases where the client acts on behalf of the resource owner, the authorization code grant is recommended. If the client acts on behalf of the resource owner but does not have any display or has very limited interaction possibilities, it is recommended to use the device code grant defined in [RFC8628]. In cases where the client acts autonomously, the client credentials grant is recommended.

For details on the different grant types, see Section 1.3 of [RFC6749]. The OAuth 2.0 framework provides an extension mechanism for defining additional grant types, so profiles of this framework **MAY** define additional grant types, if needed.

5.5. Client Credentials

Authentication of the client is mandatory independent of the grant type when requesting an access token from the token endpoint. In the case of the client credentials grant type, the authentication and grant coincide.

Client registration and provisioning of client credentials to the client is out of scope for this specification.

The OAuth framework defines one client credential type in Section 2.3.1 of [RFC6749] that comprises the client_id and client_secret values. [OAUTH-RPCC] adds raw public key and pre-shared key to the client credentials type. Profiles of this framework **MAY** extend it with an additional client credentials type using client certificates.

5.6. AS Authentication

The client credentials grant does not, by default, authenticate the AS that the client connects to. In classic OAuth, the AS is authenticated with a TLS server certificate.

Profiles of this framework **MUST** specify how clients authenticate the AS and how communication security is implemented. By default, server side TLS certificates, as defined by OAuth 2.0, are required.

5.7. The Authorization Endpoint

The OAuth 2.0 authorization endpoint is used to interact with the resource owner and obtain an authorization grant in certain grant flows. The primary use case for the ACE-OAuth framework is for machine-to-machine interactions that do not involve the resource owner in the authorization flow; therefore, this endpoint is out of scope here. Future profiles may define constrained adaptation mechanisms for this endpoint as well. Nonconstrained clients interacting with constrained resource servers can use the specification in Section 3.1 of [RFC6749] and the attack countermeasures suggested in Section 4.2 of [RFC6819].

5.8. The Token Endpoint

In standard OAuth 2.0, the AS provides the token endpoint for submitting access token requests. This framework extends the functionality of the token endpoint, giving the AS the possibility to help the client and RS establish shared keys or exchange their public keys. Furthermore, this framework defines encodings using CBOR as a substitute for JSON.

The endpoint may also be exposed over HTTPS, as in classical OAuth or even other transports. A profile **MUST** define the details of the mapping between the fields described below and these transports. If HTTPS with JSON is used, the semantics of Sections 4.1.3 and 4.1.4 of the OAuth 2.0 specification [RFC6749] **MUST** be followed (with additions as described below). If CBOR is used as the payload format, the semantics described in this section **MUST** be followed.

For the AS to be able to issue a token, the client **MUST** be authenticated and present a valid grant for the scopes requested. Profiles of this framework **MUST** specify how the AS authenticates the client and how the communication between the client and AS is protected, fulfilling the requirements specified in Section 5.

The default name of this endpoint in a url-path **SHOULD** be '/token'. However, implementations are not required to use this name and can define their own instead.

5.8.1. Client-to-AS Request

The client sends a POST request to the token endpoint at the AS. The profile **MUST** specify how the communication is protected. The content of the request consists of the parameters specified in the relevant subsection of Section 4 of the OAuth 2.0 specification [RFC6749], depending on the grant type, with the following exceptions and additions:

- The `grant_type` parameter is **OPTIONAL** in the context of this framework (as opposed to **REQUIRED** in [RFC6749]). If that parameter is missing, the default value "client_credentials" is implied.
- The `audience` parameter from [RFC8693] is **OPTIONAL** to request an access token bound to a specific audience.
- The `cnonce` parameter defined in Section 5.8.4.4 is **REQUIRED** if the RS provided a client-nonce in the AS Request Creation Hints message (Section 5.3).
- The `scope` parameter **MAY** be encoded as a byte string instead of the string encoding specified in Section 3.3 of [RFC6749] or in order to allow compact encoding of complex scopes. The syntax of such a binary encoding is explicitly not specified here and left to profiles or applications. Note specifically that a binary encoded scope does not necessarily use the space character '0x20' to delimit scope-tokens.
- The client can send an empty (null value) `ace_profile` parameter to indicate that it wants the AS to include the `ace_profile` parameter in the response. See Section 5.8.4.3.
- A client **MUST** be able to use the parameters from [RFC9201] in an access token request to the token endpoint, and the AS **MUST** be able to process these additional parameters.

The default behavior is that the AS generates a symmetric proof-of-possession key for the client. In order to use an asymmetric key pair or to reuse a key previously established with the RS, the client is supposed to use the `req_cnf` parameter from [RFC9201].

If CoAP is used, then these parameters **MUST** be provided in a CBOR map (see Table 5).

When HTTP is used as a transport, then the client makes a request to the token endpoint; the parameters **MUST** be encoded as defined in Appendix B of [RFC6749].

The following examples illustrate different types of requests for proof-of-possession tokens.

Figure 4 shows a request for a token with a symmetric proof-of-possession key, using diagnostic notation.

```
Header: POST (Code=0.02)
Uri-Host: "as.example.com"
Uri-Path: "token"
Content-Format: application/ace+cbor
Payload:
{
  / client_id / 24 : "myclient",
  / audience /  5  : "tempSensor4711"
}
```

Figure 4: Example Request for an Access Token Bound to a Symmetric Key

Figure 5 shows a request for a token with an asymmetric proof-of-possession key. Note that, in this example, OSCORE [RFC8613] is used to provide object-security; therefore, the Content-Format is "application/oscore" wrapping the "application/ace+cbor" type content. The OSCORE option has a decoded interpretation appended in parentheses for the reader's convenience. Also note that, in this example, the audience is implicitly known by both the client and AS. Furthermore, note that this example uses the `req_cnf` parameter from [RFC9201].

```
Header: POST (Code=0.02)
Uri-Host: "as.example.com"
Uri-Path: "token"
OSCORE: 0x09, 0x05, 0x44, 0x6C
  (h=0, k=1, n=001, partialIV= 0x05, kid=[0x44, 0x6C])
Content-Format: application/oscore
Payload:
  0x44025d1/ ... (full payload omitted for brevity) ... /68b3825e

Decrypted payload:
{
  / client_id / 24 : "myclient",
  / req_cnf / 4 : {
    / COSE_Key / 1 : {
      / kty /  1 : 2 / EC2 /,
      / kid /  2 : h'11',
      / crv / -1 : 1 / P-256 /,
      / x /   -2 : b64'usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8',
      / y /   -3 : b64'IBOL+C3BttVivg+lSreASjpkttcsz+1rb7btKLv8EX4'
    }
  }
}
```

Figure 5: Example Token Request Bound to an Asymmetric Key

Figure 6 shows a request for a token where a previously communicated proof-of-possession key is only referenced using the `req_cnf` parameter from [RFC9201].

```
Header: POST (Code=0.02)
Uri-Host: "as.example.com"
Uri-Path: "token"
Content-Format: application/ace+cbor
Payload:
{
  / client_id / 24 : "myclient",
  / audience /   5 : "valve424",
  / scope /      9 : "read",
  / req_cnf /    4 : {
     / kid /        3 : b64'6kg0dXJM13U'
  }
}
```

Figure 6: Example Request for an Access Token Bound to a Key Reference

Refresh tokens are typically not stored as securely as proof-of-possession keys in requesting clients. Proof-of-possession-based refresh token requests **MUST NOT** request different proof-of-possession keys or different audiences in token requests. Refresh token requests can only be used to request access tokens bound to the same proof-of-possession key and the same audience as access tokens issued in the initial token request.

5.8.2. AS-to-Client Response

If the access token request has been successfully verified by the AS and the client is authorized to obtain an access token corresponding to its access token request, the AS sends a response with the response code equivalent to the CoAP response code 2.01 (Created). If the client request was invalid, or not authorized, the AS returns an error response, as described in Section 5.8.3.

Note that the AS decides which token type and profile to use when issuing a successful response. It is assumed that the AS has prior knowledge of the capabilities of the client and the RS (see Appendix D). This prior knowledge may, for example, be set by the use of a dynamic client registration protocol exchange [RFC7591]. If the client has requested a specific proof-of-possession key using the `req_cnf` parameter from [RFC9201], this may also influence which profile the AS selects, as it needs to support the use of the key type requested by the client.

The content of the successful reply is the Access Information. When using CoAP, the payload **MUST** be encoded as a CBOR map; when using HTTP, the encoding is a JSON map, as specified in Section 5.1 of [RFC6749]. In both cases, the parameters specified in Section 5.1 of [RFC6749] are used, with the following additions and changes:

ace_profile:
: This parameter is **OPTIONAL** unless the request included an empty `ace_profile` parameter, in which case it is MANDATORY. This indicates the profile that the client **MUST** use towards the RS. See Section 5.8.4.3 for the formatting of this parameter. If this parameter is absent, the AS assumes that the client implicitly knows which profile to use towards the RS.

`token_type`:
: This parameter is **OPTIONAL**, as opposed to **REQUIRED** in [RFC6749]. By default, implementations of this framework **SHOULD** assume that the `token_type` is "PoP". If a specific use case requires another `token_type` (e.g., "Bearer") to be used, then this parameter is **REQUIRED**.

Furthermore, [RFC9201] defines additional parameters that the AS **MUST** be able to use when responding to a request to the token endpoint.

Table 2 summarizes the parameters that can currently be part of the Access Information. Future extensions may define additional parameters.

Parameter name	Specified in
`access_token`	[RFC6749]
`token_type`	[RFC6749]
`expires_in`	[RFC6749]
`refresh_token`	[RFC6749]
`scope`	[RFC6749]
`state`	[RFC6749]
`error`	[RFC6749]
`error_description`	[RFC6749]
`error_uri`	[RFC6749]
`ace_profile`	RFC 9200
`cnf`	[RFC9201]
`rs_cnf`	[RFC9201]

Table 2: Access Information Parameters

Figure 7 shows a response containing a token and a `cnf` parameter with a symmetric proof-of-possession key, which is defined in [RFC9201]. Note that the key identifier `kid` is only used to simplify indexing and retrieving the key, and no assumptions should be made that it is unique in the domains of either the client or the RS.

```
Header: Created (Code=2.01)
Content-Format: application/ace+cbor
Payload:
{
  / access_token / 1 : b64'SlAV32hk'/ ...
   (remainder of CWT omitted for brevity;
   CWT contains COSE_Key in the cnf claim)/,
  / ace_profile / 38 : "coap_dtls",
  / expires_in /   2 : 3600,
  / cnf / 8 : {
    / COSE_Key / 1 : {
      / kty / 1 : 4 / Symmetric /,
      / kid / 2 : b64'39Gqlw',
      / k /  -1 : b64'hJtXhkV8FJG+Onbc6mxC'
    }
  }
}
```

Figure 7: Example AS Response with an Access Token Bound to a Symmetric Key

5.8.3. Error Response

The error responses for interactions with the AS are generally equivalent to the ones defined in Section 5.2 of [RFC6749], with the following exceptions:

- When using CoAP, the payload **MUST** be encoded as a CBOR map, with the Content-Format "application/ace+cbor". When using HTTP, the payload is encoded in JSON, as specified in Section 5.2 of [RFC6749].
- A response code equivalent to the CoAP code 4.00 (Bad Request) **MUST** be used for all error responses, except for invalid_client, where a response code equivalent to the CoAP code 4.01 (Unauthorized) **MAY** be used under the same conditions as specified in Section 5.2 of [RFC6749].
- The parameters `error`, `error_description`, and `error_uri` **MUST** be abbreviated using the codes specified in Table 5, when a CBOR encoding is used.
- The error code (i.e., value of the `error` parameter) **MUST** be abbreviated, as specified in Table 3, when a CBOR encoding is used.

Name	CBOR Values	Original Specification
`invalid_request`	1	Section 5.2 of [RFC6749]
`invalid_client`	2	Section 5.2 of [RFC6749]
`invalid_grant`	3	Section 5.2 of [RFC6749]
`unauthorized_client`	4	Section 5.2 of [RFC6749]
`unsupported_grant_type`	5	Section 5.2 of [RFC6749]

Name	CBOR Values	Original Specification
`invalid_scope`	6	Section 5.2 of [RFC6749]
`unsupported_pop_key`	7	RFC 9200
`incompatible_ace_profiles`	8	RFC 9200

Table 3: CBOR Abbreviations for Common Error Codes

In addition to the error responses defined in OAuth 2.0, the following behavior **MUST** be implemented by the AS:

- If the client submits an asymmetric key in the token request that the RS cannot process, the AS **MUST** reject that request with a response code equivalent to the CoAP code 4.00 (Bad Request), including the error code "unsupported_pop_key" specified in Table 3.
- If the client and the RS it has requested an access token for do not share a common profile, the AS **MUST** reject that request with a response code equivalent to the CoAP code 4.00 (Bad Request), including the error code "incompatible_ace_profiles" specified in Table 3.

5.8.4. Request and Response Parameters

This section provides more detail about the new parameters that can be used in access token requests and responses, as well as abbreviations for more compact encoding of existing parameters and common parameter values.

5.8.4.1. Grant Type

The abbreviations specified in the registry defined in Section 8.5 **MUST** be used in CBOR encodings instead of the string values defined in [RFC6749] if CBOR payloads are used.

Name	CBOR Value	Original Specification
`password`	0	Section 4.3.2 of [RFC6749]
`authorization_code`	1	Section 4.1.3 of [RFC6749]
`client_credentials`	2	Section 4.4.2 of [RFC6749]
`refresh_token`	3	Section 6 of [RFC6749]

Table 4: CBOR Abbreviations for Common Grant Types

5.8.4.2. Token Type

The `token_type` parameter, defined in Section 5.1 of [RFC6749], allows the AS to indicate to the client which type of access token it is receiving (e.g., a bearer token).

This document registers the new value "PoP" for the "OAuth Access Token Types" registry, specifying a proof-of-possession token. How the proof of possession by the client to the RS is performed **MUST** be specified by the profiles.

The values in the `token_type` parameter **MUST** use the CBOR abbreviations defined in the registry specified by Section 8.7 if a CBOR encoding is used.

In this framework, the "pop" value for the `token_type` parameter is the default. The AS may, however, provide a different value from those registered in [IANA.OAuthAccessTokenTypes].

5.8.4.3. Profile

Profiles of this framework **MUST** define the communication protocol and the communication security protocol between the client and the RS. The security protocol **MUST** provide encryption, integrity, and replay protection. It **MUST** also provide a binding between requests and responses. Furthermore, profiles **MUST** define a list of allowed proof-of-possession methods if they support proof-of-possession tokens.

A profile **MUST** specify an identifier that **MUST** be used to uniquely identify itself in the `ace_profile` parameter. The textual representation of the profile identifier is intended for human readability and for JSON-based interactions; it **MUST NOT** be used for CBOR-based interactions. Profiles **MUST** register their identifier in the registry defined in Section 8.8.

Profiles **MAY** define additional parameters for both the token request and the Access Information in the access token response in order to support negotiation or signaling of profile-specific parameters.

Clients that want the AS to provide them with the `ace_profile` parameter in the access token response can indicate that by sending an `ace_profile` parameter with a null value for CBOR-based interactions, or an empty string if CBOR is not used, in the access token request.

5.8.4.4. Client-Nonce

This parameter **MUST** be sent from the client to the AS if it previously received a `cnonce` parameter in the AS Request Creation Hints (Section 5.3). The parameter is encoded as a byte string for CBOR-based interactions and as a string (base64url without padding encoded binary [RFC4648]) if CBOR is not used. It **MUST** copy the value from the `cnonce` parameter in the AS Request Creation Hints.

5.8.5. Mapping Parameters to CBOR

If CBOR encoding is used, all OAuth parameters in access token requests and responses **MUST** be mapped to CBOR types, as specified in the registry defined by Section 8.10, using the given integer abbreviation for the map keys.

Note that we have aligned the abbreviations corresponding to claims with the abbreviations defined in [RFC8392].

Note also that abbreviations from -24 to 23 have a 1-byte encoding size in CBOR. We have thus chosen to assign abbreviations in that range to parameters we expect to be used most frequently in constrained scenarios.

Name	CBOR Key	Value Type	Original Specification
access_token	1	byte string	[RFC6749]
expires_in	2	unsigned integer	[RFC6749]
audience	5	text string	[RFC8693]
scope	9	text or byte string	[RFC6749]
client_id	24	text string	[RFC6749]
client_secret	25	byte string	[RFC6749]
response_type	26	text string	[RFC6749]
redirect_uri	27	text string	[RFC6749]
state	28	text string	[RFC6749]
code	29	byte string	[RFC6749]
error	30	integer	[RFC6749]
error_description	31	text string	[RFC6749]
error_uri	32	text string	[RFC6749]
grant_type	33	unsigned integer	[RFC6749]
token_type	34	integer	[RFC6749]
username	35	text string	[RFC6749]
password	36	text string	[RFC6749]
refresh_token	37	byte string	[RFC6749]
ace_profile	38	integer	RFC 9200
cnonce	39	byte string	RFC 9200

Table 5: CBOR Mappings Used in Token Requests and Responses

5.9. The Introspection Endpoint

Token introspection [RFC7662] **MAY** be implemented by the AS and the RS. When implemented, it **MAY** be used by the RS and to query the AS for metadata about a given token, e.g., validity or scope. Analogous to the protocol defined in [RFC7662] for HTTP and JSON, this section defines adaptations to more constrained environments using CBOR and leaving the choice of the application protocol to the profile. The client **MAY** also implement and use introspection analogously to the RS to obtain information about a given token.

Communication between the requesting entity and the introspection endpoint at the AS **MUST** be integrity protected and encrypted. The communication security protocol **MUST** also provide a binding between requests and responses. Furthermore, the two interacting parties **MUST** perform mutual authentication. Finally, the AS **SHOULD** verify that the requesting entity has the right to access introspection information about the provided token. Profiles of this framework that support introspection **MUST** specify how authentication and communication security between the requesting entity and the AS is implemented.

The default name of this endpoint in a url-path **SHOULD** be '/introspect'. However, implementations are not required to use this name and can define their own instead.

5.9.1. Introspection Request

The requesting entity sends a POST request to the introspection endpoint at the AS. The profile **MUST** specify how the communication is protected. If CoAP is used, the payload **MUST** be encoded as a CBOR map with a `token` entry containing the access token. Further optional parameters representing additional context that is known by the requesting entity to aid the AS in its response **MAY** be included.

For CoAP-based interaction, all messages **MUST** use the content type "application/ace+cbor". For HTTP, the encoding defined in Section 2.1 of [RFC7662] is used.

The same parameters are required and optional as in Section 2.1 of [RFC7662].

For example, Figure 8 shows an RS calling the token introspection endpoint at the AS to query about an OAuth 2.0 proof-of-possession token. Note that object security based on OSCORE [RFC8613] is assumed in this example; therefore, the Content-Format is "application/oscore". Figure 9 shows the decoded payload.

```
Header: POST (Code=0.02)
Uri-Host: "as.example.com"
Uri-Path: "introspect"
OSCORE: 0x09, 0x05, 0x25
Content-Format: application/oscore
Payload:
... COSE content ...
```

Figure 8: Example Introspection Request

```
{
  / token / 11   : b64'7gj0dXJQ43U',
  / token_type_hint / 33 : 2 / PoP /
}
```

Figure 9: Decoded Payload

5.9.2. Introspection Response

If the introspection request is authorized and successfully processed, the AS sends a response with the response code equivalent to the CoAP code 2.01 (Created). If the introspection request was invalid, not authorized, or couldn't be processed, the AS returns an error response, as described in Section 5.9.3.

In a successful response, the AS encodes the response parameters in a map. If CoAP is used, this **MUST** be encoded as a CBOR map; if HTTP is used, the JSON encoding specified in Section 2.2 of [RFC7662] is used. The map containing the response payload includes the same required and optional parameters as in Section 2.2 of [RFC7662], with the following additions:

`ace_profile`
: This parameter is **OPTIONAL**. This indicates the profile that the RS **MUST** use with the client. See Section 5.8.4.3 for more details on the formatting of this parameter. If this parameter is absent, the AS assumes that the RS implicitly knows which profile to use towards the client.

`cnonce`
: This parameter is **OPTIONAL**. This is a client-nonce provided to the AS by the client. The RS **MUST** verify that this corresponds to the client-nonce previously provided to the client in the AS Request Creation Hints. See Sections 5.3 and 5.8.4.4. Its value is a byte string when encoded in CBOR and is the base64url encoding of this byte string without padding when encoded in JSON [RFC4648].

`cti`
: This parameter is **OPTIONAL**. This is the `cti` claim associated to this access token. This parameter has the same meaning and processing rules as the `jti` parameter defined in Section 3.1.2 of [RFC7662] except that its value is a byte string when encoded in CBOR and is the base64url encoding of this byte string without padding when encoded in JSON [RFC4648].

`exi`
: This parameter is **OPTIONAL**. This is the `expires_in` claim associated to this access token. See Section 5.10.3.

Furthermore, [RFC9201] defines more parameters that the AS **MUST** be able to use when responding to a request to the introspection endpoint.

For example, Figure 10 shows an AS response to the introspection request in Figure 8. Note that this example contains the `cnf` parameter defined in [RFC9201].

```
Header: Created (Code=2.01)
Content-Format: application/ace+cbor
Payload:
{
  / active /      10 : true,
  / scope /        9 : "read",
  / ace_profile / 38 : 1 / coap_dtls /,
  / cnf /          8 : {
    / COSE_Key / 1 : {
      / kty / 1 : 4 / Symmetric /,
      / kid / 2 : b64'39Gqlw',
      / k /  -1 : b64'hJtXhkV8FJG+Onbc6mxC'
    }
  }
}
```

Figure 10: Example Introspection Response

5.9.3. Error Response

The error responses for CoAP-based interactions with the AS are equivalent to the ones for HTTP-based interactions, as defined in Section 2.3 of [RFC7662], with the following differences:

- If content is sent and CoAP is used, the payload **MUST** be encoded as a CBOR map and the Content-Format "application/ace+cbor" **MUST** be used. For HTTP, the encoding defined in Section 2.3 of [RFC6749] is used.
- If the credentials used by the requesting entity (usually the RS) are invalid, the AS **MUST** respond with the response code equivalent to the CoAP code 4.01 (Unauthorized) and use the required and optional parameters from Section 2.3 of [RFC7662].
- If the requesting entity does not have the right to perform this introspection request, the AS **MUST** respond with a response code equivalent to the CoAP code 4.03 (Forbidden). In this case, no payload is returned.
- The parameters `error`, `error_description`, and `error_uri` **MUST** be abbreviated using the codes specified in Table 5.
- The error codes **MUST** be abbreviated using the codes specified in the registry defined by Section 8.4.

Note that a properly formed and authorized query for an inactive or otherwise invalid token does not warrant an error response by this specification. In these cases, the authorization server **MUST** instead respond with an introspection response with the `active` field set to "false".

5.9.4. Mapping Introspection Parameters to CBOR

If CBOR is used, the introspection request and response parameters **MUST** be mapped to CBOR types, as specified in the registry defined by Section 8.12, using the given integer abbreviation for the map key.

Note that we have aligned abbreviations that correspond to a claim with the abbreviations defined in [RFC8392] and the abbreviations of parameters with the same name from Section 5.8.5.

Parameter name	CBOR Key	Value Type	Original Specification
`iss`	1	text string	[RFC7662]
`sub`	2	text string	[RFC7662]
`aud`	3	text string	[RFC7662]
`exp`	4	integer or floating-point number	[RFC7662]
`nbf`	5	integer or floating-point number	[RFC7662]
`iat`	6	integer or floating-point number	[RFC7662]
`cti`	7	byte string	RFC 9200
`scope`	9	text or byte string	[RFC7662]
`active`	10	True or False	[RFC7662]
`token`	11	byte string	[RFC7662]
`client_id`	24	text string	[RFC7662]
`error`	30	integer	[RFC7662]
`error_description`	31	text string	[RFC7662]
`error_uri`	32	text string	[RFC7662]
`token_type_hint`	33	text string	[RFC7662]

Parameter name	CBOR Key	Value Type	Original Specification
token_type	34	integer	[RFC7662]
username	35	text string	[RFC7662]
ace_profile	38	integer	RFC 9200
cnonce	39	byte string	RFC 9200
exi	40	unsigned integer	RFC 9200

Table 6: CBOR Mappings for Token Introspection Parameters

5.10. The Access Token

In this framework, the use of CBOR Web Token (CWT) as specified in [RFC8392] is **RECOMMENDED**.

In order to facilitate offline processing of access tokens, this document uses the `cnf` claim from [RFC8747] and the `scope` claim from [RFC8693] for JWT- and CWT-encoded tokens. In addition to string encoding specified for the `scope` claim, a binary encoding **MAY** be used. The syntax of such an encoding is explicitly not specified here and left to profiles or applications, specifically note that a binary encoded scope does not necessarily use the space character '0x20' to delimit scope-tokens.

If the AS needs to convey a hint to the RS about which profile it should use to communicate with the client, the AS **MAY** include an `ace_profile` claim in the access token, with the same syntax and semantics as defined in Section 5.8.4.3.

If the client submitted a `cnonce` parameter in the access token request (Section 5.8.4.4), the AS **MUST** include the value of this parameter in the `cnonce` claim specified here. The `cnonce` claim uses binary encoding.

5.10.1. The Authorization Information Endpoint

The access token, containing authorization information and information about the proof-of-possession method used by the client, needs to be transported to the RS so that the RS can authenticate and authorize the client request.

This section defines a method for transporting the access token to the RS using a RESTful protocol, such as CoAP. Profiles of this framework **MAY** define other methods for token transport.

The method consists of an authz-info endpoint, implemented by the RS. A client using this method **MUST** make a POST request to the authz-info endpoint at the RS with the access token in the payload. The CoAP Content-Format or HTTP media type **MUST** reflect the format of the token, e.g., "application/cwt", for CBOR Web Tokens; if no Content-Format or media type is defined for the token format, "application/octet-stream" **MUST** be used.

The RS receiving the token **MUST** verify the validity of the token. If the token is valid, the RS **MUST** respond to the POST request with a response code equivalent to CoAP code 2.01 (Created). Section 5.10.1.1 outlines how an RS **MUST** proceed to verify the validity of an access token.

The RS **MUST** be prepared to store at least one access token for future use. This is a difference as to how access tokens are handled in OAuth 2.0, where the access token is typically sent along with each request and therefore not stored at the RS.

When using this framework, it is **RECOMMENDED** that an RS stores only one token per proof-of-possession key. This means that an additional token linked to the same key will supersede any existing token at the RS by replacing the corresponding authorization information. The reason is that this greatly simplifies (constrained) implementations, with respect to required storage and resolving a request to the applicable token. The use of multiple access tokens for a single client increases the strain on the resource server, as it must consider every access token and calculate the actual permissions of the client. Also, tokens may contradict each other, which may lead the server to enforce wrong permissions. If one of the access tokens expires earlier than others, the resulting permissions may offer insufficient protection.

If the payload sent to the authz-info endpoint does not parse to a token, the RS **MUST** respond with a response code equivalent to the CoAP code 4.00 (Bad Request).

The RS **MAY** make an introspection request to validate the token before responding to the POST request to the authz-info endpoint, e.g., if the token is an opaque reference. Some transport protocols may provide a way to indicate that the RS is busy and the client should retry after an interval; this type of status update would be appropriate while the RS is waiting for an introspection response.

Profiles **MUST** specify whether the authz-info endpoint is protected, including whether error responses from this endpoint are protected. Note that since the token contains information that allows the client and the RS to establish a security context in the first place, mutual authentication may not be possible at this point.

The default name of this endpoint in a url-path is '/authz-info'; however, implementations are not required to use this name and can define their own instead.

5.10.1.1. Verifying an Access Token

When an RS receives an access token, it **MUST** verify it before storing it. The details of token verification depends on various aspects, including the token encoding, the type of token, the security protection applied to the token, and the claims. The token encoding matters since the security protection differs between the token encodings. For example, a CWT token uses COSE, while a JWT token uses JSON Object Signing and Encryption (JOSE). The type of token also has an influence on the verification procedure since tokens may be self-contained, whereby token verification may happen locally at the RS, while a reference token requires further interaction with the authorization server, for example, using token introspection, to obtain the claims associated with the token reference. Self-contained tokens **MUST** at least be integrity protected, but they **MAY** also be encrypted.

For self-contained tokens, the RS **MUST** process the security protection of the token first, as specified by the respective token format. For CWT, the description can be found in [RFC8392]; for JWT, the relevant specification is [RFC7519]. This **MUST** include a verification that security protection (and thus the token) was generated by an AS that has the right to issue access tokens for this RS.

In case the token is communicated by reference, the RS needs to obtain the claims first. When the RS uses token introspection, the relevant specification is [RFC7662] with CoAP transport specified in Section 5.9.

Errors may happen during this initial processing stage:

- If the verification of the security wrapper fails, or the token was issued by an AS that does not have the right to issue tokens for the receiving RS, the RS **MUST** discard the token and, if this was an interaction with authz-info, return an error message with a response code equivalent to the CoAP code 4.01 (Unauthorized).
- If the claims cannot be obtained, the RS **MUST** discard the token and, in case of an interaction via the authz-info endpoint, return an error message with a response code equivalent to the CoAP code 4.00 (Bad Request).

Next, the RS **MUST** verify claims, if present, contained in the access token. Errors are returned when claim checks fail, in the order of priority of this list:

`iss`
: The `iss` claim (if present) must identify the AS that has produced the security protection for the access token. If that is not the case, the RS **MUST** discard the token. If this was an interaction with authz-info, the RS **MUST** also respond with a response code equivalent to the CoAP code 4.01 (Unauthorized).

`exp`
: The expiration date must be in the future. If that is not the case, the RS **MUST** discard the token. If this was an interaction with authz-info, the RS **MUST** also respond with a response code equivalent to the CoAP code 4.01 (Unauthorized). Note that the RS has to terminate access rights to the protected resources at the time when the tokens expire.

`aud`
: The `aud` claim must refer to an audience that the RS identifies with. If that is not the case, the RS **MUST** discard the token. If this was an interaction with authz-info, the RS **MUST** also respond with a response code equivalent to the CoAP code 4.03 (Forbidden).

`scope`
: The RS must recognize value of the `scope` claim. If that is not the case, the RS **MUST** discard the token. If this was an interaction with authz-info, the RS **MUST** also respond with a response code equivalent to the CoAP code 4.00 (Bad Request). The RS **MAY** provide additional information in the error response to clarify what went wrong.

Additional processing may be needed for other claims in a way specific to a profile or the underlying application.

Note that the `sub` (Subject) claim cannot always be verified when the token is submitted to the RS since the client may not have authenticated yet. Also note that a counter for the `exi` (expires in) claim **MUST** be initialized when the RS first verifies this token.

Also note that profiles of this framework may define access token transport mechanisms that do not allow for error responses. Therefore, the error messages specified here only apply if the token was sent to the authz-info endpoint.

When sending error responses, the RS **MAY** use the error codes from Section 3.1 of [RFC6750] to provide additional details to the client.

5.10.1.2. Protecting the Authorization Information Endpoint

As this framework can be used in RESTful environments, it is important to make sure that attackers cannot perform unauthorized requests on the authz-info endpoints, other than submitting access tokens.

Specifically, it **SHOULD NOT** be possible to perform GET, DELETE, or PUT on the authz-info endpoint.

The RS **SHOULD** implement rate-limiting measures to mitigate attacks aiming to overload the processing capacity of the RS by repeatedly submitting tokens. For CoAP-based communication, the RS could use the mechanisms from [RFC8516] to indicate that it is overloaded.

5.10.2. Client Requests to the RS

Before sending a request to an RS, the client **MUST** verify that the keys used to protect this communication are still valid. See Section 5.10.4 for details on how the client determines the validity of the keys used.

If an RS receives a request from a client and the target resource requires authorization, the RS **MUST** first verify that it has an access token that authorizes this request and that the client has performed the proof-of-possession binding for that token to the request.

The response code **MUST** be 4.01 (Unauthorized) in case the client has not performed the proof of possession or if the RS has no valid access token for the client. If the RS has an access token for the client but the token does not authorize access for the resource that was requested, the RS **MUST** reject the request with a 4.03 (Forbidden). If the RS has an access token for the client but it does not cover the action that was requested on the resource, the RS **MUST** reject the request with a 4.05 (Method Not Allowed).

Note: The use of the response codes 4.03 and 4.05 is intended to prevent infinite loops where a client optimistically tries to access a requested resource with any access token received from AS. As malicious clients could pretend to be the C to determine the C's privileges, these detailed response codes must be used only when a certain level of security is already available, which can be achieved only when the client is authenticated.

Note: The RS **MAY** use introspection for timely validation of an access token at the time when a request is presented.

Note: Matching the claims of the access token (e.g., `scope`) to a specific request is application specific.

If the request matches a valid token and the client has performed the proof of possession for that token, the RS continues to process the request as specified by the underlying application.

5.10.3. Token Expiration

Depending on the capabilities of the RS, there are various ways in which it can verify the expiration of a received access token. The following is a list of the possibilities including what functionality they require of the RS.

- The token is a CWT and includes an `exp` claim and possibly the `nbf` claim. The RS verifies these by comparing them to values from its internal clock, as defined in [RFC7519]. In this case, the RS's internal clock must reflect the current date and time or at least be synchronized with the AS's clock. How this clock synchronization would be performed is out of scope for this specification.
- The RS verifies the validity of the token by performing an introspection request, as specified in Section 5.9. This requires the RS to have a reliable network connection to the AS and to be able to handle two secure sessions in parallel (C to RS and RS to AS).
- In order to support token expiration for devices that have no reliable way of synchronizing their internal clocks, this specification defines the following approach: The claim `exi` (expires in) can be used to provide the RS with the lifetime of the token in seconds from the time the RS first receives the token. This mechanism only works for self-contained tokens, i.e., CWTs and JWTs. For CWTs, this parameter is encoded as an unsigned integer, while JWTs encode this as JSON number.
- Processing this claim requires that the RS does the following:
  - For each token the RS receives that contains an `exi` claim, keep track of the time it received that token and revisit that list regularly to expunge expired tokens.
  - Keep track of the identifiers of tokens containing the `exi` claim that have expired (in order to avoid accepting them again). In order to avoid an unbounded memory usage growth, this **MUST** be implemented in the following way when the `exi` claim is used:
    - When creating the token, the AS **MUST** add a `cti` claim (or `jti` for JWTs) to the access token. The value of this claim **MUST** be created as the binary representation of the concatenation of the identifier of the RS with a sequence number counting the tokens containing an `exi` claim, issued by this AS for the RS.
    - The RS **MUST** store the highest sequence number of an expired token containing the `exi` claim that it has seen and treat tokens with lower sequence numbers as expired. Note that this could lead to discarding valid tokens with lower sequence numbers if the AS where to issue tokens of different validity time for the same RS. The assumption is that typically tokens in such a scenario would all have the same validity time.

If a token that authorizes a long-running request, such as a CoAP Observe [RFC7641], expires, the RS **MUST** send an error response with the response code equivalent to the CoAP code 4.01 (Unauthorized) to the client and then terminate processing the long-running request.

5.10.4. Key Expiration

The AS provides the client with key material that the RS uses. This can either be a common symmetric PoP key or an asymmetric key used by the RS to authenticate towards the client. Since there is currently no expiration metadata associated to those keys, the client has no way of knowing if these keys are still valid. This may lead to situations where the client sends requests containing sensitive information to the RS using a key that is expired and possibly in the hands of an attacker or where the client accepts responses from the RS that are not properly protected and could possibly have been forged by an attacker.

In order to prevent this, the client must assume that those keys are only valid as long as the related access token is. Since the access token is opaque to the client, one of the following methods **MUST** be used to inform the client about the validity of an access token:

- The client knows a default validity time for all tokens it is using (i.e., how long a token is valid after being issued). This information could be provisioned to the client when it is registered at the AS or published by the AS in a way that the client can query.
- The AS informs the client about the token validity using the `expires_in` parameter in the Access Information.

A client that is not able to obtain information about the expiration of a token **MUST NOT** use this token.

6. Security Considerations

Security considerations applicable to authentication and authorization in RESTful environments provided in OAuth 2.0 [RFC6749] apply to this work. Furthermore, [RFC6819] provides additional security considerations for OAuth, which apply to IoT deployments as well. If the introspection endpoint is used, the security considerations from [RFC7662] also apply.

The following subsections address issues specific to this document and its use in constrained environments.

6.1. Protecting Tokens

A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature or a keyed message digest, e.g., a Message Authentication Code (MAC) or an Authenticated Encryption with Associated Data (AEAD) algorithm. Consequently, the token integrity protection **MUST** be applied to prevent the token from being modified, particularly since it contains a reference to the symmetric key or the asymmetric key used for proof of possession. If the access token contains the symmetric key, this symmetric key **MUST** be encrypted by the authorization server so that only the resource server can decrypt it. Note that using an AEAD algorithm is preferable over using a MAC unless the token needs to be publicly readable.

If the token is intended for multiple recipients (i.e., an audience that is a group), integrity protection of the token with a symmetric key, shared between the AS and the recipients, is not sufficient, since any of the recipients could modify the token undetected by the other recipients. Therefore, a token with a multirecipient audience **MUST** be protected with an asymmetric signature.

It is important for the authorization server to include the identity of the intended recipient (the audience), typically a single resource server (or a list of resource servers), in the token. The same shared secret **MUST NOT** be used as a proof-of-possession key with multiple resource servers, since the benefit from using the proof-of-possession concept is then significantly reduced.

If clients are capable of doing so, they should frequently request fresh access tokens, as this allows the AS to keep the lifetime of the tokens short. This allows the AS to use shorter proof-of-possession key sizes, which translate to a performance benefit for the client and for the resource server. Shorter keys also lead to shorter messages (particularly with asymmetric keying material).

When authorization servers bind symmetric keys to access tokens, they **SHOULD** scope these access tokens to a specific permission.

In certain situations, it may be necessary to revoke an access token that is still valid. Client-initiated revocation is specified in [RFC7009] for OAuth 2.0. Other revocation mechanisms are currently not specified, as the underlying assumption in OAuth is that access tokens are issued with a relatively short lifetime. This may not hold true for disconnected constrained devices needing access tokens with relatively long lifetimes and would therefore necessitate further standardization work that is out of scope for this document.

6.2. Communication Security

Communication with the authorization server **MUST** use confidentiality protection. This step is extremely important since the client or the RS may obtain the proof-of-possession key from the authorization server for use with a specific access token. Not using confidentiality protection exposes this secret (and the access token) to an eavesdropper, thereby completely negating proof-of-possession security. The requirements for communication security of profiles are specified in Section 5.

Additional protection for the access token can be applied by encrypting it, for example, encryption of CWTs is specified in Section 7.1 of [RFC8392]. Such additional protection can be necessary if the token is later transferred over an insecure connection (e.g., when it is sent to the authz-info endpoint).

Care must be taken by developers to prevent leakage of the PoP credentials (i.e., the private key or the symmetric key). An adversary in possession of the PoP credentials bound to the access token will be able to impersonate the client. Be aware that this is a real risk with many constrained environments, since adversaries may get physical access to the devices and can

therefore use physical extraction techniques to gain access to memory contents. This risk can be mitigated to some extent by making sure that keys are refreshed frequently, by using software isolation techniques, and by using hardware security.

6.3. Long-Term Credentials

Both the clients and RSs have long-term credentials that are used to secure communications and authenticate to the AS. These credentials need to be protected against unauthorized access. In constrained devices deployed in publicly accessible places, such protection can be difficult to achieve without specialized hardware (e.g., secure key storage memory).

If credentials are lost or compromised, the operator of the affected devices needs to have procedures to invalidate any access these credentials give and needs to revoke tokens linked to such credentials. The loss of a credential linked to a specific device **MUST NOT** lead to a compromise of other credentials not linked to that device; therefore, secret keys used for authentication **MUST NOT** be shared between more than two parties.

Operators of the clients or RSs **SHOULD** have procedures in place to replace credentials that are suspected to have been compromised or that have been lost.

Operators also **SHOULD** have procedures for decommissioning devices that include securely erasing credentials and other security-critical material in the devices being decommissioned.

6.4. Unprotected AS Request Creation Hints

Initially, no secure channel exists to protect the communication between the C and RS. Thus, the C cannot determine if the AS Request Creation Hints contained in an unprotected response from the RS to an unauthorized request (see Section 5.3) are authentic. Therefore, the C **MUST** determine if an AS is authorized to provide access tokens for a certain RS. How this determination is implemented is out of scope for this document and left to the applications.

6.5. Minimal Security Requirements for Communication

This section summarizes the minimal requirements for the communication security of the different protocol interactions.

C-AS
: All communication between the client and the authorization server **MUST** be encrypted and integrity and replay protected. Furthermore, responses from the AS to the client **MUST** be bound to the client's request to avoid attacks where the attacker swaps the intended response for an older one valid for a previous request. This requires that the client and the authorization server have previously exchanged either a shared secret or their public keys in order to negotiate a secure communication. Furthermore, the client **MUST** be able to determine whether an AS has the authority to issue access tokens for a certain RS. This can, for example, be done through preconfigured lists or through an online lookup mechanism that in turn also must be secured.

RS-AS
: The communication between the resource server and the authorization server via the introspection endpoint **MUST** be encrypted and integrity and replay protected. Furthermore, responses from the AS to the RS **MUST** be bound to the RS's request. This requires that the RS and the authorization server have previously exchanged either a shared secret or their public keys in order to negotiate a secure communication. Furthermore, the RS **MUST** be able to determine whether an AS has the authority to issue access tokens itself. This is usually configured out of band but could also be performed through an online lookup mechanism, provided that it is also secured in the same way.

C-RS
: The initial communication between the client and the resource server cannot be secured in general, since the RS is not in possession of on access token for that client, which would carry the necessary parameters. If both parties support DTLS without client authentication, it is **RECOMMENDED** to use this mechanism for protecting the initial communication. After the client has successfully transmitted the access token to the RS, a secure communication protocol **MUST** be established between the client and RS for the actual resource request. This protocol **MUST** provide confidentiality, integrity, and replay protection, as well as a binding between requests and responses. This requires that the client learned either the RS's public key or received a symmetric proof-of-possession key bound to the access token from the AS. The RS must have learned either the client's public key, a shared symmetric key from the claims in the token, or an introspection request. Since ACE does not provide profile negotiation between the C and RS, the client **MUST** have learned what profile the RS supports (e.g., from the AS or preconfigured) and initiated the communication accordingly.

6.6. Token Freshness and Expiration

An RS that is offline faces the problem of clock drift. Since it cannot synchronize its clock with the AS, it may be tricked into accepting old access tokens that are no longer valid or have been compromised. In order to prevent this, an RS may use the nonce-based mechanism (`cnonce`) defined in Section 5.3 to ensure freshness of an Access Token subsequently presented to this RS.

Another problem with clock drift is that evaluating the standard token expiration claim `exp` can give unpredictable results.

Acceptable ranges of clock drift are highly dependent on the concrete application. Important factors are how long access tokens are valid and how critical timely expiration of the access token is.

The expiration mechanism implemented by the `exi` claim, based on the first time the RS sees the token, was defined to provide a more predictable alternative. The `exi` approach has some drawbacks that need to be considered:

- A malicious client may hold back tokens with the `exi` claim in order to prolong their lifespan.
- If an RS loses state (e.g., due to an unscheduled reboot), it may lose the current values of counters tracking the `exi` claims of tokens it is storing.

The first drawback is inherent to the deployment scenario and the `exi` solution. It can therefore not be mitigated without requiring the RS be online at times. The second drawback can be mitigated by regularly storing the value of `exi` counters to persistent memory.

6.7. Combining Profiles

There may be use cases where different transport and security protocols are allowed for the different interactions, and, if that is not explicitly covered by an existing profile, it corresponds to combining profiles into a new one. For example, a new profile could specify that a previously defined MQTT-TLS profile is used between the client and the RS in combination with a previously defined CoAP-DTLS profile for interactions between the client and the AS. The new profile that combines existing profiles **MUST** specify how the existing profiles' security requirements remain satisfied. Therefore, any profile **MUST** clearly specify its security requirements and **MUST** document if its security depends on the combination of various protocol interactions.

6.8. Unprotected Information

Communication with the authz-info endpoint, as well as the various error responses defined in this framework, potentially includes sending information over an unprotected channel. These messages may leak information to an adversary or may be manipulated by active attackers to induce incorrect behavior. For example, error responses for requests to the authorization information endpoint can reveal information about an otherwise opaque access token to an adversary who has intercepted this token.

As far as error messages are concerned, this framework is written under the assumption that, in general, the benefits of detailed error messages outweigh the risk due to information leakage. For particular use cases where this assessment does not apply, detailed error messages can be replaced by more generic ones.

In some scenarios, it may be possible to protect the communication with the authz-info endpoint (e.g., through DTLS with only server-side authentication). In cases where this is not possible, it is **RECOMMENDED** to use encrypted CWTs or tokens that are opaque references and need to be subjected to introspection by the RS.

If the initial Unauthorized Resource Request message (see Section 5.2) is used, the client **MUST** make sure that it is not sending sensitive content in this request. While GET and DELETE requests only reveal the target URI of the resource, POST and PUT requests would reveal the whole payload of the intended operation.

Since the client is not authenticated at the point when it is submitting an access token to the authz-info endpoint, attackers may be pretending to be a client and trying to trick an RS to use an obsolete profile that in turn specifies a vulnerable security mechanism via the authz-info endpoint. Such an attack would require a valid access token containing an `ace_profile` claim requesting the use of said obsolete profile. Resource owners should update the configuration of their RSs to prevent them from using such obsolete profiles.

6.9. Identifying Audiences

The `aud` claim, as defined in [RFC7519], and the equivalent `audience` parameter from [RFC8693] are intentionally vague on how to match the audience value to a specific RS. This is intended to allow application-specific semantics to be used. This section attempts to give some general guidance for the use of audiences in constrained environments.

URLs are not a good way of identifying mobile devices that can switch networks and thus be associated with new URLs. If the audience represents a single RS and asymmetric keys are used, the RS can be uniquely identified by a hash of its public key. If this approach is used, it is **RECOMMENDED** to apply the procedure from Section 3 of [RFC6920].

If the audience addresses a group of resource servers, the mapping of a group identifier to an individual RS has to be provisioned to each RS before the group-audience is usable. Managing dynamic groups could be an issue if any RS is not always reachable when the groups' memberships change. Furthermore, issuing access tokens bound to symmetric proof-of-possession keys that apply to a group-audience is problematic, as an RS that is in possession of the access token can impersonate the client towards the other RSs that are part of the group. It is therefore **NOT RECOMMENDED** to issue access tokens bound to a group-audience and symmetric proof-of possession keys.

Even the client must be able to determine the correct values to put into the `audience` parameter in order to obtain a token for the intended RS. Errors in this process can lead to the client inadvertently obtaining a token for the wrong RS. The correct values for `audience` can either be provisioned to the client as part of its configuration or dynamically looked up by the client in some directory. In the latter case, the integrity and correctness of the directory data must be assured. Note that the `audience` hint provided by the RS as part of the AS Request Creation Hints (Section 5.3) is not typically source authenticated and integrity protected and should therefore not be treated a trusted value.

6.10. Denial of Service Against or with Introspection

The optional introspection mechanism provided by OAuth and supported in the ACE framework allows for two types of attacks that need to be considered by implementers.

First, an attacker could perform a denial-of-service attack against the introspection endpoint at the AS in order to prevent validation of access tokens. To maintain the security of the system, an RS that is configured to use introspection **MUST NOT** allow access based on a token for which it couldn't reach the introspection endpoint.

Second, an attacker could use the fact that an RS performs introspection to perform a denial-of-service attack against that RS by repeatedly sending tokens to its authz-info endpoint that require an introspection call. The RS can mitigate such attacks by implementing rate limits on how many introspection requests they perform in a given time interval for a certain client IP address

submitting tokens to /authz-info. When that limit has been reached, incoming requests from that address are rejected for a certain amount of time. A general rate limit on the introspection requests should also be considered in order to mitigate distributed attacks.

7. Privacy Considerations

Implementers and users should be aware of the privacy implications of the different possible deployments of this framework.

The AS is in a very central position and can potentially learn sensitive information about the clients requesting access tokens. If the client credentials grant is used, the AS can track what kind of access the client intends to perform. With other grants, this can be prevented by the resource owner. To do so, the resource owner needs to bind the grants it issues to anonymous, ephemeral credentials that do not allow the AS to link different grants and thus different access token requests by the same client.

The claims contained in a token can reveal privacy-sensitive information about the client and the RS to any party having access to them (whether by processing the content of a self-contained token or by introspection). The AS **SHOULD** be configured to minimize the information about clients and RSs disclosed in the tokens it issues.

If tokens are only integrity protected and not encrypted, they may reveal information to attackers listening on the wire or be able to acquire the access tokens in some other way. In the case of CWTs, the token may, e.g., reveal the audience, the scope, and the confirmation method used by the client. The latter may reveal the identity of the device or application running the client. This may be linkable to the identity of the person using the client (if there is a person and not a machine-to-machine interaction).

Clients using asymmetric keys for proof of possession should be aware of the consequences of using the same key pair for proof of possession towards different RSs. A set of colluding RSs or an attacker able to obtain the access tokens will be able to link the requests or even to determine the client's identity.

An unprotected response to an unauthorized request (see Section 5.3) may disclose information about the RS and/or its existing relationship with the C. It is advisable to include as little information as possible in an unencrypted response. Even the absolute URI of the AS may reveal sensitive information about the service that the RS provides. Developers must ensure that the RS does not disclose information that has an impact on the privacy of the stakeholders in the AS Request Creation Hints. They may choose to use a different mechanism for the discovery of the AS if necessary. If means of encrypting communication between the C and RS already exist, more detailed information may be included with an error response to provide the C with sufficient information to react on that particular error.

8. IANA Considerations

This document creates several registries with a registration policy of Expert Review; guidelines to the experts are given in Section 8.17.

8.1. ACE Authorization Server Request Creation Hints

This specification establishes the IANA "ACE Authorization Server Request Creation Hints" registry.

The columns of the registry are:

Name: The name of the parameter.

CBOR Key: CBOR map key for the parameter. Different ranges of values use different registration policies [RFC8126]. Integer values from -256 to 255 are designated as Standards Action. Integer values from -65536 to -257 and from 256 to 65535 are designated as Specification Required. Integer values greater than 65535 are designated as Expert Review. Integer values less than -65536 are marked as Private Use.

Value Type: The CBOR data types allowable for the values of this parameter.

Reference: This contains a pointer to the public specification of the Request Creation Hint abbreviation, if one exists.

This registry has been initially populated by the values in Table 1. The Reference column for all of these entries is this document.

8.2. CoRE Resource Types

IANA has registered a new Resource Type (rt=) Link Target Attribute in the "Resource Type (rt=) Link Target Attribute Values" subregistry under the "Constrained RESTful Environments (CoRE) Parameters" [IANA.CoreParameters] registry:

Value: `ace.ai`
Description: ACE-OAuth authz-info endpoint resource.
Reference: RFC 9200

Specific ACE-OAuth profiles can use this common resource type for defining their profile-specific discovery processes.

8.3. OAuth Extensions Errors

This specification registers the following error values in the "OAuth Extensions Error Registry" [IANA.OAuthExtensionsErrorRegistry].

Name: `unsupported_pop_key`
Usage Location: token error response
Protocol Extension: RFC 9200
Change Controller: IETF
Reference: Section 5.8.3 of RFC 9200

Name: `incompatible_ace_profiles`
Usage Location: token error response
Protocol Extension: RFC 9200
Change Controller: IETF
Reference: Section 5.8.3 of RFC 9200

8.4. OAuth Error Code CBOR Mappings

This specification establishes the IANA "OAuth Error Code CBOR Mappings" registry.

The columns of the registry are:

Name: The OAuth Error Code name, refers to the name in Section 5.2 of [RFC6749], e.g., "invalid_request".

CBOR Value: CBOR abbreviation for this error code. Integer values less than -65536 are marked as Private Use; all other values use the registration policy Expert Review [RFC8126].

Reference: This contains a pointer to the public specification of the error code abbreviation, if one exists.

Original Specification: This contains a pointer to the public specification of the error code, if one exists.

This registry has been initially populated by the values in Table 3. The Reference column for all of these entries is this document.

8.5. OAuth Grant Type CBOR Mappings

This specification establishes the IANA "OAuth Grant Type CBOR Mappings" registry.

The columns of this registry are:

Name: The name of the grant type, as specified in Section 1.3 of [RFC6749].

CBOR Value: CBOR abbreviation for this grant type. Integer values less than -65536 are marked as Private Use; all other values use the registration policy Expert Review [RFC8126].

Reference: This contains a pointer to the public specification of the grant type abbreviation, if one exists.

Original Specification: This contains a pointer to the public specification of the grant type, if one exists.

This registry has been initially populated by the values in Table 4. The Reference column for all of these entries is this document.

8.6. OAuth Access Token Types

This section registers the following new token type in the "OAuth Access Token Types" registry [IANA.OAuthAccessTokenTypes].

Name: `PoP`
Additional Token Endpoint Response Parameters: `cnf`, `rs_cnf` (see Section 3.1 of [RFC8747] and Section 3.2 of [RFC9201]).
HTTP Authentication Scheme(s): N/A
Change Controller: IETF
Reference: RFC 9200

8.7. OAuth Access Token Type CBOR Mappings

This specification establishes the IANA "OAuth Access Token Type CBOR Mappings" registry.

The columns of this registry are:

Name: The name of the token type, as registered in the "OAuth Access Token Types" registry, e.g., "Bearer".

CBOR Value: CBOR abbreviation for this token type. Integer values less than -65536 are marked as Private Use; all other values use the registration policy Expert Review [RFC8126].

Reference: This contains a pointer to the public specification of the OAuth token type abbreviation, if one exists.

Original Specification: This contains a pointer to the public specification of the OAuth token type, if one exists.

8.7.1. Initial Registry Contents

Name: `Bearer`
CBOR Value: 1
Reference: RFC 9200
Original Specification: [RFC6749]

Name: `PoP`
CBOR Value: 2
Reference: RFC 9200
Original Specification: RFC 9200

8.8. ACE Profiles

This specification establishes the IANA "ACE Profile" registry.

The columns of this registry are:

Name: The name of the profile to be used as the value of the profile attribute.

Description: Text giving an overview of the profile and the context it is developed for.

CBOR Value: CBOR abbreviation for this profile name. Different ranges of values use different registration policies [RFC8126]. Integer values from -256 to 255 are designated as Standards Action. Integer values from -65536 to -257 and from 256 to 65535 are designated as Specification Required. Integer values greater than 65535 are designated as Expert Review. Integer values less than -65536 are marked as Private Use.

Reference: This contains a pointer to the public specification of the profile abbreviation, if one exists.

8.9. OAuth Parameters

This specification registers the following parameter in the "OAuth Parameters" registry [IANA.OAuthParameters]:

Name: `ace_profile`
Parameter Usage Location: token response
Change Controller: IETF
Reference: Sections 5.8.2 and 5.8.4.3 of RFC 9200

8.10. OAuth Parameters CBOR Mappings

This specification establishes the IANA "OAuth Parameters CBOR Mappings" registry.

The columns of this registry are:

Name: The OAuth Parameter name, refers to the name in the OAuth parameter registry, e.g., `client_id`.

CBOR Key: CBOR map key for this parameter. Integer values less than -65536 are marked as Private Use; all other values use the registration policy Expert Review [RFC8126].

Value Type: The allowable CBOR data types for values of this parameter.

Reference: This contains a pointer to the public specification of the OAuth parameter abbreviation, if one exists.

Original Specification This contains a pointer to the public specification of the OAuth parameter, if one exists.

This registry has been initially populated by the values in Table 5. The Reference column for all of these entries is this document.

8.11. OAuth Introspection Response Parameters

This specification registers the following parameters in the "OAuth Token Introspection Response" registry [IANA.TokenIntrospectionResponse].

Name: `ace_profile`
Description: The ACE profile used between the client and RS.
Change Controller: IETF
Reference: Section 5.9.2 of RFC 9200

Name: `cnonce`
Description: "client-nonce". A nonce previously provided to the AS by the RS via the client. Used to verify token freshness when the RS cannot synchronize its clock with the AS.
Change Controller: IETF
Reference: Section 5.9.2 of RFC 9200

Name `cti`
Description "CWT ID". The identifier of a CWT as defined in [RFC8392].
Change Controller IETF
Reference Section 5.9.2 of RFC 9200

Name: `exi`
Description: "Expires in". Lifetime of the token in seconds from the time the RS first sees it. Used to implement a weaker form of token expiration for devices that cannot synchronize their internal clocks.
Change Controller: IETF
Reference: Section 5.9.2 of RFC 9200

8.12. OAuth Token Introspection Response CBOR Mappings

This specification establishes the IANA "OAuth Token Introspection Response CBOR Mappings" registry.

The columns of this registry are:

Name: The OAuth Parameter name, refers to the name in the OAuth parameter registry, e.g., `client_id`.

CBOR Key: CBOR map key for this parameter. Integer values less than -65536 are marked as Private Use; all other values use the registration policy Expert Review [RFC8126].

Value Type: The allowable CBOR data types for values of this parameter.

Reference: This contains a pointer to the public specification of the introspection response parameter abbreviation, if one exists.

Original Specification This contains a pointer to the public specification of the OAuth Token Introspection parameter, if one exists.

This registry has been initially populated by the values in Table 6. The Reference column for all of these entries is this document.

Note that the mappings of parameters corresponding to claim names intentionally coincide with the CWT claim name mappings from [RFC8392].

8.13. JSON Web Token Claims

This specification registers the following new claims in the "JSON Web Token Claims" subregistry under the "JSON Web Token (JWT)" registry [IANA.JsonWebTokenClaims]:

Claim Name: `ace_profile`
Claim Description: The ACE profile a token is supposed to be used with.
Change Controller: IETF
Reference: Section 5.10 of RFC 9200

Claim Name: `cnonce`
Claim Description: "client-nonce". A nonce previously provided to the AS by the RS via the client. Used to verify token freshness when the RS cannot synchronize its clock with the AS.
Change Controller: IETF
Reference: Section 5.10 of RFC 9200

Claim Name: `exi`
Claim Description: "Expires in". Lifetime of the token in seconds from the time the RS first sees it. Used to implement a weaker form of token expiration for devices that cannot synchronize their internal clocks.
Change Controller: IETF
Reference: Section 5.10.3 of RFC 9200

8.14. CBOR Web Token Claims

This specification registers the following new claims in the "CBOR Web Token (CWT) Claims" registry [IANA.CborWebTokenClaims].

Claim Name: `ace_profile`

Claim Description: The ACE profile a token is supposed to be used with.
JWT Claim Name: `ace_profile`
Claim Key: 38
Claim Value Type: integer
Change Controller: IETF
Reference: Section 5.10 of RFC 9200

Claim Name: `cnonce`
Claim Description: The client-nonce sent to the AS by the RS via the client.
JWT Claim Name: `cnonce`
Claim Key: 39
Claim Value Type: byte string
Change Controller: IETF
Reference: Section 5.10 of RFC 9200

Claim Name: `exi`
Claim Description: The expiration time of a token measured from when it was received at the RS in seconds.
JWT Claim Name: `exi`
Claim Key: 40
Claim Value Type: unsigned integer
Change Controller: IETF
Reference: Section 5.10.3 of RFC 9200

Claim Name: `scope`
Claim Description: The scope of an access token, as defined in [RFC6749].
JWT Claim Name: `scope`
Claim Key: 9
Claim Value Type: byte string or text string
Change Controller: IETF
Reference: Section 4.2 of [RFC8693]

8.15. Media Type Registration

This specification registers the "application/ace+cbor" media type for messages of the protocols defined in this document carrying parameters encoded in CBOR. This registration follows the procedures specified in [RFC6838].

Type name: application

Subtype name: ace+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: Must be encoded as a CBOR map containing the protocol parameters defined in RFC 9200.

Security considerations: See Section 6 of RFC 9200

Interoperability considerations: N/A

Published specification: RFC 9200

Applications that use this media type: The type is used by authorization servers, clients, and resource servers that support the ACE framework with CBOR encoding, as specified in RFC 9200.

Fragment identifier considerations: N/A

Additional information: N/A

Person & email address to contact for further information:
IESG <iesg@ietf.org>

Intended usage: COMMON

Restrictions on usage: none

Author: Ludwig Seitz <ludwig.seitz@combitech.se>

Change controller: IETF

8.16. CoAP Content-Formats

The following entry has been registered in the "CoAP Content-Formats" registry:

Media Type: application/ace+cbor
Encoding: -
ID: 19
Reference: RFC 9200

8.17. Expert Review Instructions

All of the IANA registries established in this document are defined to use a registration policy of Expert Review. This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason, so they should be given substantial latitude.

Expert Reviewers should take into consideration the following points:

- Point squatting should be discouraged. Reviewers are encouraged to get sufficient information for registration requests to ensure that the usage is not going to duplicate one that is already registered and that the point is likely to be used in deployments. The zones tagged as Private Use are intended for testing purposes and closed environments; code points in other ranges should not be assigned for testing.

- Specifications are needed for the first-come, first-serve range if they are expected to be used outside of closed environments in an interoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.
- Experts should take into account the expected usage of fields when approving point assignment. The fact that there is a range for Standards Track documents does not mean that a Standards Track document cannot have points assigned outside of that range. The length of the encoded value should be weighed against how many code points of that length are left, i.e., the size of device it will be used on.
- Since a high degree of overlap is expected between these registries and the contents of the OAuth parameters [IANA.OAuthParameters] registries, experts should require new registrations to maintain alignment with parameters from OAuth that have comparable functionality. Deviation from this alignment should only be allowed if there are functional differences that are motivated by the use case and that cannot be easily or efficiently addressed by comparable OAuth parameters.

9. References

9.1. Normative References

[IANA.CborWebTokenClaims] IANA, "CBOR Web Token (CWT) Claims", <https://www.iana.org/assignments/cwt>.

[IANA.CoreParameters] IANA, "Constrained RESTful Environments (CoRE) Parameters", <https://www.iana.org/assignments/core-parameters>.

[IANA.JsonWebTokenClaims] IANA, "JSON Web Token Claims", <https://www.iana.org/assignments/jwt>.

[IANA.OAuthAccessTokenTypes] IANA, "OAuth Access Token Types", <https://www.iana.org/assignments/oauth-parameters>.

[IANA.OAuthExtensionsErrorRegistry] IANA, "OAuth Extensions Error Registry", <https://www.iana.org/assignments/oauth-parameters>.

[IANA.OAuthParameters] IANA, "OAuth Parameters", <https://www.iana.org/assignments/oauth-parameters>.

[IANA.TokenIntrospectionResponse] IANA, "OAuth Token Introspection Response", <https://www.iana.org/assignments/oauth-parameters>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", RFC 6920, DOI 10.17487/RFC6920, April 2013, <https://www.rfc-editor.org/info/rfc6920>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <https://www.rfc-editor.org/info/rfc7662>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/info/rfc8693>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9201] Seitz, L., "Additional OAuth Parameters for Authentication and Authorization in Constrained Environments (ACE)", RFC 9201, DOI 10.17487/RFC9201, August 2022, <https://www.rfc-editor.org/info/rfc9201>.

9.2. Informative References

[BLE] Bluetooth Special Interest Group, "Core Specification 5.3", Section 4.4, July 2021, <https://www.bluetooth.com/specifications/bluetooth-core-specification/>.

[DCAF] Gerdes, S., Bergmann, O., and C. Bormann, "Delegated CoAP Authentication and Authorization Framework (DCAF)", Work in Progress, Internet-Draft, draft-gerdes-ace-dcaf-authorize-04, 19 October 2015, <https://datatracker.ietf.org/doc/html/draft-gerdes-ace-dcaf-authorize-04>.

[Margi10impact] Margi, C., de Oliveira, B., de Sousa, G., Simplicio Jr, M., Barreto, P., Carvalho, T., Naeslund, M., and R. Gold, "Impact of Operating Systems on Wireless Sensor Networks (Security) Applications and Testbeds", Proceedings of the 19th International Conference on Computer Communications and Networks, DOI 10.1109/ICCCN.2010.5560028, August 2010, <https://doi.org/10.1109/ICCCN.2010.5560028>.

[MQTT5.0] Banks, A., Briggs, E., Borgendale, K., and R. Gupta, "MQTT Version 5.0", OASIS Standard, March 2019, <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>.

[OAUTH-RPCC] Seitz, L., Erdtman, S., and M. Tiloca, "Raw-Public-Key and Pre-Shared-Key as OAuth client credentials", Work in Progress, Internet-Draft, draft-erdtman-oauth-rpcc-00, 21 November 2017, <https://datatracker.ietf.org/doc/html/draft-erdtman-oauth-rpcc-00>.

[POP-KEY-DIST] Bradley, J., Hunt, P., Jones, M., Tschofenig, H., and M. Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", Work in Progress, Internet-Draft, draft-ietf-oauth-pop-key-distribution-07, 27 March 2019, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-pop-key-distribution-07>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <https://www.rfc-editor.org/info/rfc6690>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/info/rfc7009>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <https://www.rfc-editor.org/info/rfc7641>.

[RFC7744] Seitz, L., Ed., Gerdes, S., Ed., Selander, G., Mani, M., and S. Kumar, "Use Cases for Authentication and Authorization in Constrained Environments", RFC 7744, DOI 10.17487/RFC7744, January 2016, <https://www.rfc-editor.org/info/rfc7744>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <https://www.rfc-editor.org/info/rfc7959>.

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8516] Keranen, A., ""Too Many Requests" Response Code for the Constrained Application Protocol", RFC 8516, DOI 10.17487/RFC8516, January 2019, <https://www.rfc-editor.org/info/rfc8516>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/info/rfc8613>.

[RFC8628] Denniss, W., Bradley, J., Jones, M., and H. Tschofenig, "OAuth 2.0 Device Authorization Grant", RFC 8628, DOI 10.17487/RFC8628, August 2019, <https://www.rfc-editor.org/info/rfc8628>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9113] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/info/rfc9113>.

[RFC9147] Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, DOI 10.17487/RFC9147, April 2022, <https://www.rfc-editor.org/info/rfc9147>.

[RFC9202] Gerdes, S., Bergmann, O., Bormann, C., Selander, G., and L. Seitz, "Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE)", RFC 9202, DOI 10.17487/RFC9202, August 2022, <https://www.rfc-editor.org/info/rfc9202>.

[RFC9203] Palombini, F., Seitz, L., Selander, G., and M. Gunnarsson, "The Object Security for Constrained RESTful Environments (OSCORE) Profile of the Authentication and Authorization for Constrained Environments (ACE) Framework", RFC 9203, DOI 10.17487/RFC9203, August 2022, <https://www.rfc-editor.org/info/rfc9203>.

Appendix A. Design Justification

This section provides further insight into the design decisions of the solution documented in this document. Section 3 lists several building blocks and briefly summarizes their importance. The justification for offering some of those building blocks, as opposed to using OAuth 2.0 as is, is given below.

Common IoT constraints are:

Low Power Radio:
: Many IoT devices are equipped with a small battery that needs to last for a long time. For many constrained wireless devices, the highest energy cost is associated to transmitting or receiving messages (roughly by a factor of 10 compared to AES) [Margi10impact]. It is therefore important to keep the total communication overhead low, including minimizing the number and size of messages sent and received, which has an impact of choice on the message format and protocol. By using CoAP over UDP and CBOR-encoded messages, some of these aspects are addressed. Security protocols contribute to the communication overhead and can, in some cases, be optimized. For example, authentication and key establishment may, in certain cases where security requirements allow, be replaced by the provisioning of security context by a trusted third party, using transport or application-layer security.

Low CPU Speed:
: Some IoT devices are equipped with processors that are significantly slower than those found in most current devices on the Internet. This typically has implications on what timely cryptographic operations a device is capable of performing, which in turn impacts, e.g., protocol latency. Symmetric key cryptography may be used instead of the computationally more expensive public key cryptography where the security requirements so allow, but this may also require support for trusted, third-party-assisted secret key establishment using transport- or application-layer security.

Small Amount of Memory:
: Microcontrollers embedded in IoT devices are often equipped with only a small amount of RAM and flash memory, which places limitations on what kind of processing can be performed and how much code can be put on those devices. To reduce code size, fewer and smaller protocol implementations can be put on the firmware of such a device. In this case, CoAP may be used instead of HTTP, symmetric-key cryptography may be used instead of public-key cryptography, and CBOR may be used instead of JSON. An authentication and key establishment protocol, e.g., the DTLS handshake, in comparison with assisted key establishment, also has an impact on memory and code footprints.

User Interface Limitations:
: Protecting access to resources is both an important security as well as privacy feature. End users and enterprise customers may not want to give access to the data collected by their IoT device or to functions it may offer to third parties. Since the classical approach of requesting permissions from end users via a rich user interface does not work in many IoT deployment scenarios, these functions need to be delegated to user-controlled devices that are better suitable for such tasks, such as smartphones and tablets.

Communication Constraints:
: In certain constrained settings, an IoT device may not be able to communicate with a given device at all times. Devices may be sleeping or just disconnected from the Internet because of general lack of connectivity in the area, cost reasons, or security reasons, e.g., to avoid an entry point for denial-of-service attacks.

The communication interactions this framework builds upon (as shown graphically in Figure 1) may be accomplished using a variety of different protocols, and not all parts of the message flow are used in all applications due to the communication constraints. Deployments making use of CoAP are expected, but this framework is not limited to them. Other protocols, such as HTTP or Bluetooth Smart communication, that do not necessarily use IP could also be used. The latter raises the need for application-layer security over the various interfaces.

In the light of these constraints, we have made the following design decisions:

CBOR, COSE, CWT:
: When using this framework, it is **RECOMMENDED** to use CBOR [RFC8949] as the data format. Where CBOR data needs to be protected, the use of COSE [RFC8152] is **RECOMMENDED**. Furthermore, where self-contained tokens are needed, it is **RECOMMENDED** to use CWT [RFC8392]. These measures aim at reducing the size of messages sent over the wire, the RAM size of data objects that need to be kept in memory, and the size of libraries that devices need to support.

CoAP:
: When using this framework, it is **RECOMMENDED** to use CoAP [RFC7252] instead of HTTP. This does not preclude the use of other protocols specifically aimed at constrained devices, e.g., Bluetooth Low Energy (see Section 3.2). This aims again at reducing the size of messages sent over the wire, the RAM size of data objects that need to be kept in memory, and the size of libraries that devices need to support.

Access Information:
: This framework defines the name "Access Information" for data concerning the RS that the AS returns to the client in an access token response (see Section 5.8.2). This aims at enabling scenarios where a powerful client supporting multiple profiles needs to interact with an RS for which it does not know the supported profiles and the raw public key.

Proof of Possession:
: This framework makes use of proof-of-possession tokens, using the `cnf` claim [RFC8747]. A request parameter `cnf` and a Response parameter `cnf`, both having a value space semantically and syntactically identical to the `cnf` claim, are defined for the token endpoint to allow requesting and stating confirmation keys. This aims at making token theft harder. Token theft is specifically relevant in constrained use cases, as communication often passes through middleboxes, which could be able to steal bearer tokens and use them to gain unauthorized access.

Authz-Info endpoint:
: This framework introduces a new way of providing access tokens to an RS by exposing an authz-info endpoint to which access tokens can be POSTed. This aims at reducing the size of the request message and the code complexity at the RS. The size of the request message is problematic, since many constrained protocols have severe message size limitations at the physical layer (e.g., in the order of 100 bytes). This means that larger packets get fragmented,

which in turn combines badly with the high rate of packet loss and the need to retransmit the whole message if one packet gets lost. Thus, separating sending of the request and sending of the access tokens helps to reduce fragmentation.

Client Credentials Grant:
: In this framework, the use of the client credentials grant is **RECOMMENDED** for machine-to-machine communication use cases, where manual intervention of the resource owner to produce a grant token is not feasible. The intention is that the resource owner would instead prearrange authorization with the AS based on the client's own credentials. The client can then (without manual intervention) obtain access tokens from the AS.

Introspection:
: In this framework, the use of access token introspection is **RECOMMENDED** in cases where the client is constrained in a way that it cannot easily obtain new access tokens (i.e., it has connectivity issues that prevent it from communicating with the AS). In that case, it is **RECOMMENDED** to use a long-term token that could be a simple reference. The RS is assumed to be able to communicate with the AS and can therefore perform introspection in order to learn the claims associated with the token reference. The advantage of such an approach is that the resource owner can change the claims associated to the token reference without having to be in contact with the client, thus granting or revoking access rights.

Appendix B. Roles and Responsibilities

Resource Owner

- Make sure that the RS is registered at the AS. This includes making known to the AS which profiles, token_type, scopes, and key types (symmetric/asymmetric) the RS supports. Also making it known to the AS which audience(s) the RS identifies itself with.
- Make sure that clients can discover the AS that is in charge of the RS.
- If the client-credentials grant is used, make sure that the AS has the necessary, up-to-date access control policies for the RS.

Requesting Party

- Make sure that the client is provisioned the necessary credentials to authenticate to the AS.
- Make sure that the client is configured to follow the security requirements of the requesting party when issuing requests (e.g., minimum communication security requirements or trust anchors).
- Register the client at the AS. This includes making known to the AS which profiles, token_types, and key types (symmetric/asymmetric) for the client.

Authorization Server

- Register the RS and manage corresponding security contexts.
- Register clients and authentication credentials.
- Allow resource owners to configure and update access control policies related to their registered RSs.

- Expose the token endpoint to allow clients to request tokens.
- Authenticate clients that wish to request a token.
- Process a token request using the authorization policies configured for the RS.
- Optionally, expose the introspection endpoint that allows RSs to submit token introspection requests.
- If providing an introspection endpoint, authenticate RSs that wish to get an introspection response.
- If providing an introspection endpoint, process token introspection requests.
- Optionally, handle token revocation.
- Optionally, provide discovery metadata. See [RFC8414].
- Optionally, handle refresh tokens.

Client

- Discover the AS in charge of the RS that is to be targeted with a request.
- Submit the token request (see step (A) of Figure 1).
  - Authenticate to the AS.
  - Optionally (if not preconfigured), specify which RS, which resource(s), and which action(s) the request(s) will target.
  - If raw public keys (RPKs) or certificates are used, make sure the AS has the right RPK or certificate for this client.
- Process the access token and Access Information (see step (B) of Figure 1).
  - Check that the Access Information provides the necessary security parameters (e.g., PoP key or information on communication security protocols supported by the RS).
  - Safely store the proof-of-possession key.
  - If provided by the AS, safely store the refresh token.
- Send the token and request to the RS (see step (C) of Figure 1).
  - Authenticate towards the RS (this could coincide with the proof-of-possession process).
  - Transmit the token as specified by the AS (default is to the authz-info endpoint; alternative options are specified by profiles).
  - Perform the proof-of-possession procedure as specified by the profile in use (this may already have been taken care of through the authentication procedure).
- Process the RS response (see step (F) of Figure 1) of the RS.

Resource Server

- Expose a way to submit access tokens. By default, this is the authz-info endpoint.
- Process an access token.
  - Verify the token is from a recognized AS.
  - Check the token's integrity.
  - Verify that the token applies to this RS.

- Check that the token has not expired (if the token provides expiration information).
- Store the token so that it can be retrieved in the context of a matching request.

Note: The order proposed here is not normative; any process that arrives at an equivalent result can be used. A noteworthy consideration is whether one can use cheap operations early on to quickly discard nonapplicable or invalid tokens before performing expensive cryptographic operations (e.g., doing an expiration check before verifying a signature).

- Process a request.
  - Set up communication security with the client.
  - Authenticate the client.
  - Match the client against existing tokens.
  - Check that tokens belonging to the client actually authorize the requested action.
  - Optionally, check that the matching tokens are still valid, using introspection (if this is possible.)
- Send a response following the agreed upon communication security mechanism(s).
- Safely store credentials, such as raw public keys, for authentication or proof-of-possession keys linked to access tokens.

Appendix C. Requirements on Profiles

This section lists the requirements on profiles of this framework for the convenience of profile designers.

- Optionally, define new methods for the client to discover the necessary permissions and AS for accessing a resource different from the one proposed in Sections 5.1 and 4
- Optionally, specify new grant types (Section 5.4).
- Optionally, define the use of client certificates as client credential type (Section 5.5).
- Specify the communication protocol the client and RS must use (e.g., CoAP) (Sections 5 and 5.8.4.3).
- Specify the security protocol the client and RS must use to protect their communication (e.g., OSCORE or DTLS). This must provide encryption and integrity and replay protection (Section 5.8.4.3).
- Specify how the client and the RS mutually authenticate (Section 4).
- Specify the proof-of-possession protocol(s) and how to select one if several are available. Also specify which key types (e.g., symmetric/asymmetric) are supported by a specific proof-of-possession protocol (Section 5.8.4.2).
- Specify a unique `ace_profile` identifier (Section 5.8.4.3).
- If introspection is supported, specify the communication and security protocol for introspection (Section 5.9).
- Specify the communication and security protocol for interactions between the client and AS. This must provide encryption, integrity protection, replay protection, and a binding between requests and responses (Sections 5 and 5.8).

- Specify how/if the authz-info endpoint is protected, including how error responses are protected (Section 5.10.1).
- Optionally, define other methods of token transport than the authz-info endpoint (Section 5.10.1).

Appendix D. Assumptions on AS Knowledge about the C and RS

This section lists the assumptions on what an AS should know about a client and an RS in order to be able to respond to requests to the token and introspection endpoints. How this information is established is out of scope for this document.

- The identifier of the client or RS.
- The profiles that the client or RS supports.
- The scopes that the RS supports.
- The audiences that the RS identifies with.
- The key types (e.g., pre-shared symmetric key, raw public key, key length, and other key parameters) that the client or RS supports.
- The types of access tokens the RS supports (e.g., CWT).
- If the RS supports CWTs, the COSE parameters for the crypto wrapper (e.g., algorithm, key-wrap algorithm, and key-length) that the RS supports.
- The expiration time for access tokens issued to this RS (unless the RS accepts a default time chosen by the AS).
- The symmetric key shared between the client and AS (if any).
- The symmetric key shared between the RS and AS (if any).
- The raw public key of the client or RS (if any).
- Whether the RS has synchronized time (and thus is able to use the `exp` claim) or not.

Appendix E. Differences to OAuth 2.0

This document adapts OAuth 2.0 to be suitable for constrained environments. This section lists the main differences from the normative requirements of OAuth 2.0.

Use of TLS
: OAuth 2.0 requires the use of TLS to protect the communication between the AS and client when requesting an access token, between the client and RS when accessing a resource, and between the AS and RS if introspection is used. This framework requires similar security properties but does not require that they be realized with TLS. See Section 5.

Cardinality of `grant_type` parameter
: In client-to-AS requests using OAuth 2.0, the `grant_type` parameter is required (per [RFC6749]). In this framework, this parameter is optional. See Section 5.8.1.

Encoding of `scope` parameter
: In client-to-AS requests using OAuth 2.0, the `scope` parameter is string encoded (per [RFC6749]). In this framework, this parameter may also be encoded as a byte string. See Section 5.8.1.

Cardinality of `token_type` parameter
: In AS-to-client responses using OAuth 2.0, the `token_type` parameter is required (per [RFC6749]). In this framework, this parameter is optional. See Section 5.8.2.

Access token retention
: In OAuth 2.0, the access token may be sent with every request to the RS. The exact use of access tokens depends on the semantics of the application and the session management concept it uses. In this framework, the RS must be able to store these tokens for later use. See Section 5.10.1.

Appendix F. Deployment Examples

There is a large variety of IoT deployments, as is indicated in Appendix A, and this section highlights a few common variants. This section is not normative but illustrates how the framework can be applied.

For each of the deployment variants, there are a number of possible security setups between clients, resource servers, and authorization servers. The main focus in the following subsections is on how authorization of a client request for a resource hosted by an RS is performed. This requires the security of the requests and responses between the clients and the RS to be considered.

Note: CBOR diagnostic notation is used for examples of requests and responses.

F.1. Local Token Validation

In this scenario, the case where the resource server is offline is considered, i.e., it is not connected to the AS at the time of the access request. This access procedure involves steps (A), (B), (C), and (F) of Figure 1.

Since the resource server must be able to verify the access token locally, self-contained access tokens must be used.

This example shows the interactions between a client, the authorization server, and a temperature sensor acting as a resource server. Message exchanges A and B are shown in Figure 11.

A: The client first generates a public-private key pair used for communication security with the RS.

The client sends a CoAP POST request to the token endpoint at the AS. The security of this request can be transport or application layer. It is up the communication security profile to define. In the example, it is assumed that both the client and AS have performed mutual authentication, e.g., via DTLS. The request contains the public key of the client and the `audience` parameter set to "tempSensorInLivingRoom", a value that the temperature sensor identifies itself with. The AS evaluates the request and authorizes the client to access the resource.

B: The AS responds with a 2.05 (Content) response containing the Access Information, including the access token. The PoP access token contains the public key of the client, and the Access Information contains the public key of the RS. For communication security, this example uses DTLS RawPublicKey between the client and the RS. The issued token will have a short validity time, i.e., `exp` close to `iat`, in order to mitigate attacks using stolen client credentials. The token includes claims, such as `scope`, with the authorized access that an owner of the temperature device can enjoy. In this example, the `scope` claim issued by the AS informs the RS that the owner of the token that can prove the possession of a key is authorized to make a GET request against the /temperature resource and a POST request on the /firmware resource. Note that the syntax and semantics of the `scope` claim are application specific.

Note: In this example, it is assumed that the client knows what resource it wants to access and is therefore able to request specific `audience` and `scope` claims for the access token.

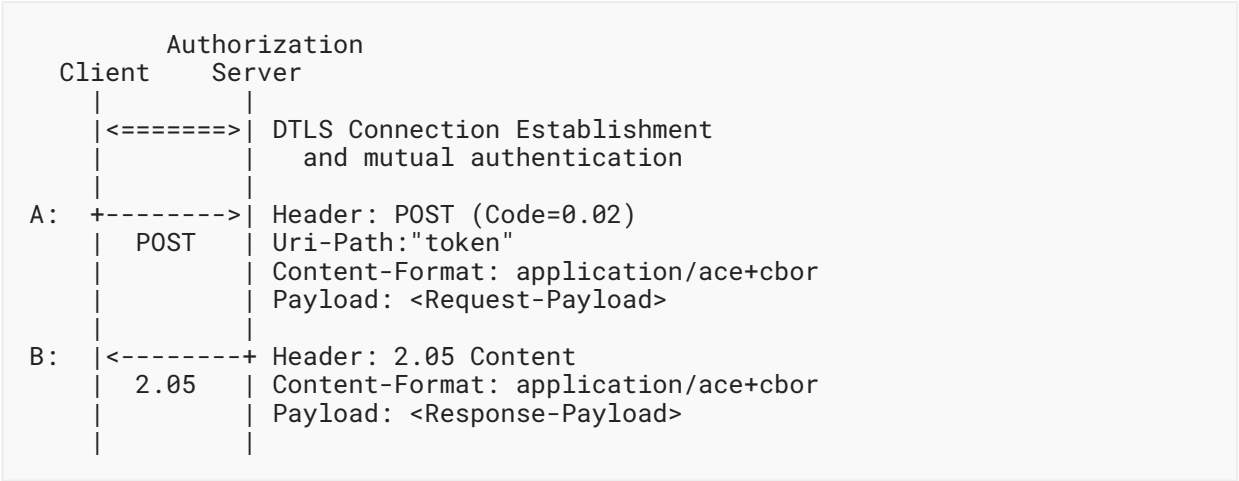

Figure 11: Token Request and Response Using Client Credentials

The information contained in the Request-Payload and the Response-Payload is shown in Figure 12. Note that the parameter `rs_cnf` from [RFC9201] is used to inform the client about the resource server's public key.

```
Request-Payload :
{
  / audience / 5 : "tempSensorInLivingRoom",
  / client_id / 24 : "myclient",
  / req_cnf / 4 : {
  / COSE_Key / 1 : {
      / kid / 2 : b64'1Bg8vub9tLe1gHMzV76e',
      / kty / 1 : 2 / EC2 /,
      / crv / -1 : 1 / P-256 /,
      / x / -2 : b64'f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU',
      / y / -3 : b64'x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0'
    }
  }
}

Response-Payload :
{
  / access_token / 1 : b64'0INDoQEKoQVNKkXfb7xaWqMT'/ .../,
  / rs_cnf / 41 : {
    / COSE_Key / 1 : {
      / kid / 2 : b64'c29tZSBwdWJsaWMga2V5IGlk',
      / kty / 1 : 2 / EC2 /,
      / crv / -1 : 1 / P-256 /,
      / x / -2   : b64'MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4',
      / y / -3   : b64'4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM'
    }
  }
}
```

Figure 12: Request and Response Payload Details

The content of the access token is shown in Figure 13.

```
{
  / aud / 3 : "tempSensorInLivingRoom",
  / iat / 6 : 1563451500,
  / exp / 4 : 1563453000,
  / scope / 9 :  "temperature_g firmware_p",
  / cnf / 8 : {
    / COSE_Key / 1 : {
      / kid / 2 : b64'1Bg8vub9tLe1gHMzV76e',
      / kty / 1 : 2 / EC2 /,
      / crv / -1 : 1 / P-256 /,
      / x / -2 : b64'f83OJ3D2xF1Bg8vub9tLe1gHMzV76e8Tus9uPHvRVEU',
      / y / -3 : b64'x_FEzRu9m36HLN_tue659LNpXW6pCyStikYjKIWI5a0'
    }
  }
}
```

Figure 13: Access Token Including Public Key of the Client

Messages C and F are shown in Figures 14 and 15.

C: The client then sends the PoP access token to the authz-info endpoint at the RS. This is a plain CoAP POST request, i.e., no transport or application-layer security is used between the client and RS since the token is integrity protected between the AS and RS. The RS verifies that the PoP access token was created by a known and trusted AS, which it applies to this RS, and that it is valid. The RS caches the security context together with authorization information about this client contained in the PoP access token.

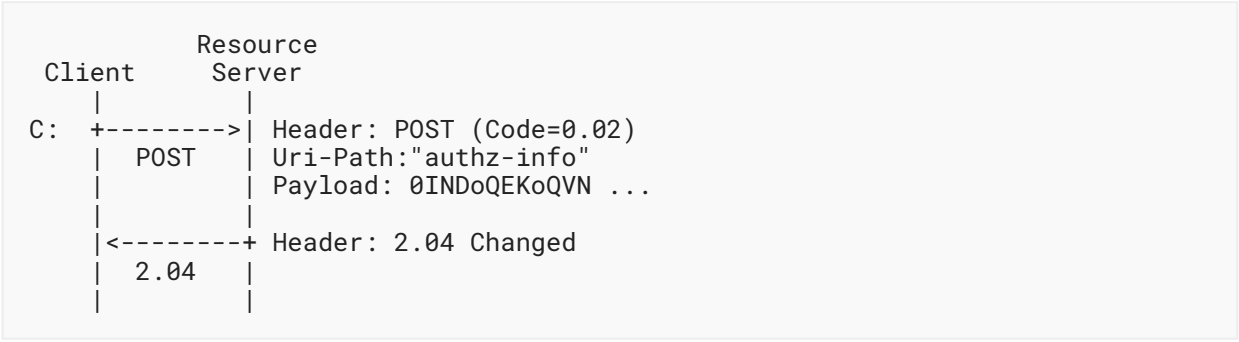

Figure 14: Access Token Provisioning to the RS

The client and the RS runs the DTLS handshake using the raw public keys established in steps B and C.

The client sends a CoAP GET request to /temperature on the RS over DTLS. The RS verifies that the request is authorized based on previously established security context.

F: The RS responds over the same DTLS channel with a CoAP 2.05 Content response containing a resource representation as payload.

Figure 15: Resource Request and Response Protected by DTLS

F.2. Introspection Aided Token Validation

In this deployment scenario, it is assumed that a client is not able to access the AS at the time of the access request, whereas the RS is assumed to be connected to the back-end infrastructure. Thus, the RS can make use of token introspection. This access procedure involves steps (A)-(F) of Figure 1 but assumes steps (A) and (B) have been carried out during a phase when the client had connectivity to the AS.

Since the client is assumed to be offline, at least for a certain period of time, a preprovisioned access token has to be long lived. Since the client is constrained, the token will not be self-contained (i.e., not a CWT) but instead just a reference. The resource server uses its connectivity to learn about the claims associated to the access token by using introspection, which is shown in the example below.

In the example, interactions between an offline client (key fob), an RS (online lock), and an AS is shown. It is assumed that there is a provisioning step where the client has access to the AS. This corresponds to message exchanges A and B, which are shown in Figure 16.

Authorization consent from the resource owner can be preconfigured, but it can also be provided via an interactive flow with the resource owner. An example of this for the key fob case could be that the resource owner has a connected car and buys a generic key to use with the car. To authorize the key fob, the owner connects it to a computer that then provides the UI for the device. After that, OAuth 2.0 implicit flow can be used to authorize the key for the car at the car manufacturer's AS.

Note: In this example, the client does not know the exact door it will be used to access since the token request is not sent at the time of access. So the `scope` and `audience` parameters are set quite wide to start with, while tailored values narrowing down the claims to the specific RS being accessed can be provided to that RS during an introspection step.

A: The client sends a CoAP POST request to the token endpoint at the AS. The request contains the `audience` parameter set to "PACS1337" (Physical Access System (PACS)), a value that identifies the physical access control system to which the individual doors are connected. The AS generates an access token as an opaque string, which it can match to the specific client and the targeted audience. It furthermore generates a symmetric proof-of-possession key. The communication security and authentication between the client and AS is assumed to have been provided at the transport layer (e.g., via DTLS) using a pre-shared security context (pre-shared key (PSK), RPK, or certificate).

B: The AS responds with a CoAP 2.05 Content response, containing as payload the Access Information, including the access token and the symmetric proof-of-possession key. Communication security between the C and RS will be DTLS and PreSharedKey. The PoP key is used as the PreSharedKey.

Note: In this example, we are using a symmetric key for a multi-RS audience, which is not recommended normally (see Section 6.9). However, in this case, the risk is deemed to be acceptable, since all the doors are part of the same physical access control system; therefore, the risk of a malicious RS impersonating the client towards another RS is low.

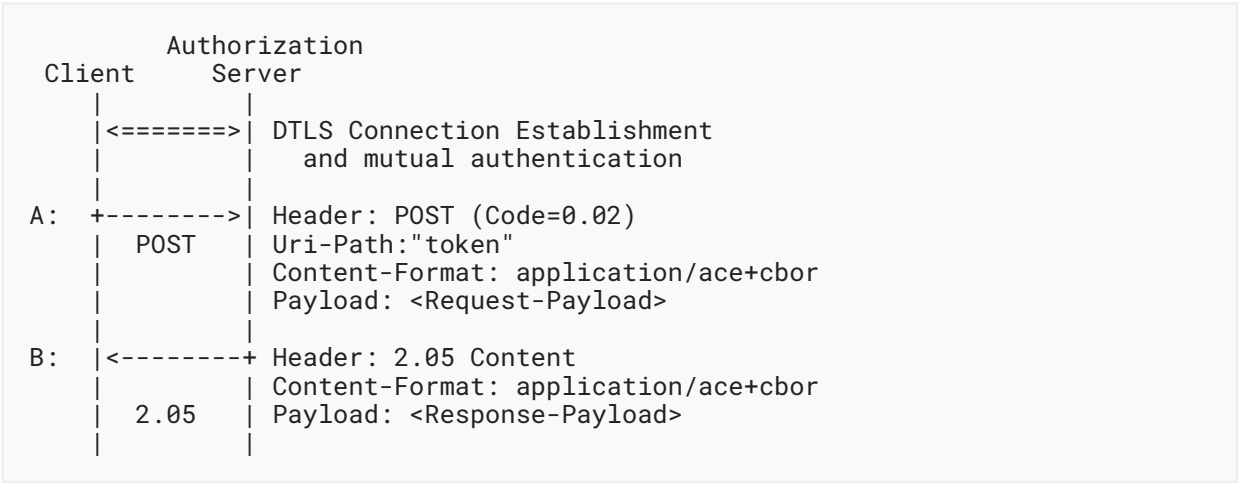

Figure 16: Token Request and Response Using Client Credentials

The information contained in the Request-Payload and the Response-Payload is shown in Figure 17.

```
Request-Payload:
{
  / client_id / 24 : "keyfob",
  / audience / 5   : "PACS1337"
}

Response-Payload:
{
  / access_token / 1 : b64'VGVzdCB0b2tlbg',
  / cnf / 8 : {
    / COSE_Key / 1 : {
      / kid / 2 : b64'c29tZSBwdWJsaWMga2V5IGlk',
      / kty / 1 : 4 / Symmetric /,
      / k / -1  : b64'ZoRSOrFzN_FzUA5XKMYoVHyzff5oRJxl-IXRtztJ6uE'
    }
  }
}
```

Figure 17: Request and Response Payload for the C Offline

In this case, the access token is just an opaque byte string referencing the authorization information at the AS.

C:

Next, the client POSTs the access token to the authz-info endpoint in the RS. This is a plain CoAP request, i.e., no DTLS between the client and RS. Since the token is an opaque string, the RS cannot verify it on its own, and thus defers to respond to the client with a status code until after step E.

D: The RS sends the token to the introspection endpoint on the AS using a CoAP POST request. In this example, the RS and AS are assumed to have performed mutual authentication using a pre-shared security context (PSK, RPK, or certificate) with the RS acting as the DTLS client.

E: The AS provides the introspection response (2.05 Content) containing parameters about the token. This includes the confirmation key (`cnf`) parameter that allows the RS to verify the client's proof of possession in step F. Note that our example in Figure 19 assumes a preestablished key (e.g., one used by the client and the RS for a previous token) that is now only referenced by its key identifier `kid`.

After receiving message E, the RS responds to the client's POST in step C with the CoAP response code 2.01 (Created).

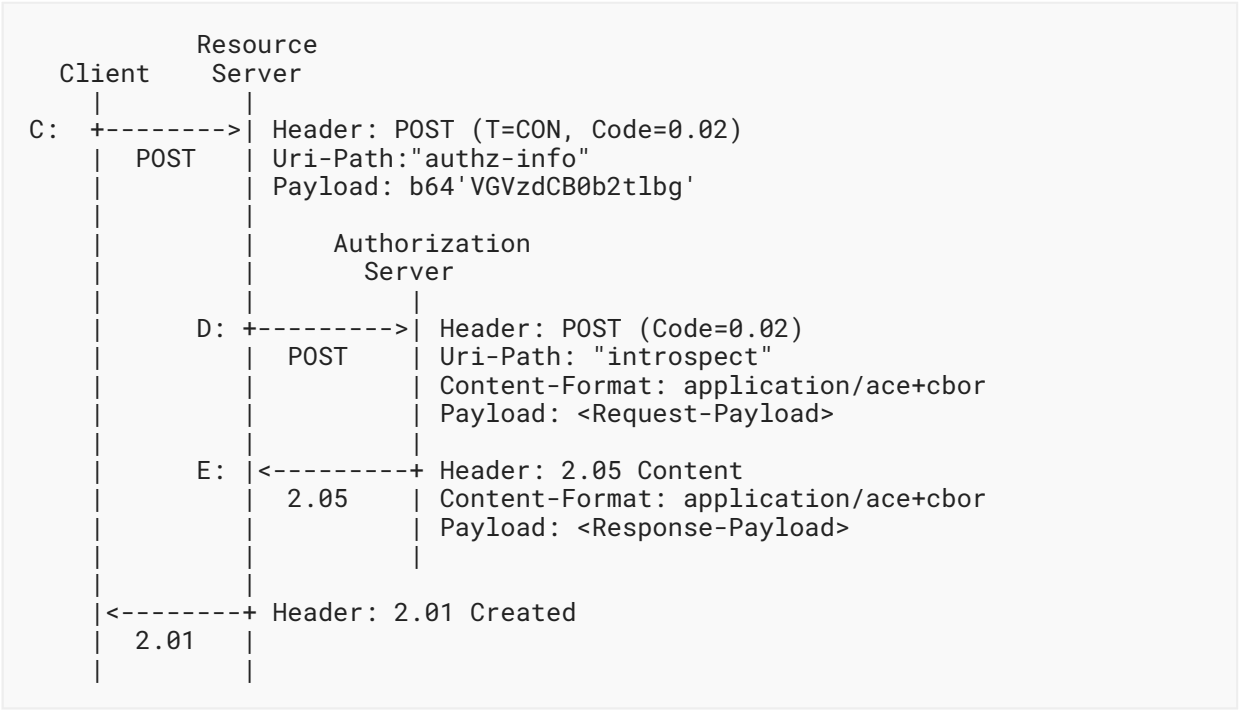

Figure 18: Token Introspection for the C Offline

The information contained in the Request-Payload and the Response-Payload is shown in Figure 19.

```
Request-Payload:
{
  / token /     11 : b64'VGVzdCB0b2tlbg',
  / client_id / 24 : "FrontDoor"
}

Response-Payload:
{
  / active / 10 : true,
  / aud /     3 : "lockOfDoor4711",
  / scope /   9 : "open close",
  / iat /     6 : 1563454000,
  / cnf /     8 : {
         / kid / 3 : b64'c29tZSBwdWJsaWMga2V5IGlk'
  }
}
```

Figure 19: Request and Response Payload for Introspection

The client uses the symmetric PoP key to establish a DTLS PreSharedKey secure connection to the RS. The CoAP request PUT is sent to the uri-path /state on the RS, changing the state of the door to locked.

F: The RS responds with an appropriate response over the secure DTLS channel.

```
          Resource
  Client    Server
    |          |
    |<========>| DTLS Connection Establishment
    |          |   using Pre Shared Key
    |          |
    +-------->| Header: PUT (Code=0.03)
    | PUT     | Uri-Path: "state"
    |         | Payload: <new state for the lock>
    |         |
F:  |<--------+ Header: 2.04 Changed
    | 2.04    | Payload: <new state for the lock>
    |         |
```

Figure 20: Resource Request and Response Protected by OSCORE

Acknowledgments

This document is a product of the ACE Working Group of the IETF.

Thanks to Eve Maler for her contributions to the use of OAuth 2.0 and Unlicensed Mobile Access (UMA) in IoT scenarios, Robert Taylor for his discussion input, and Mališa Vučinić for his input on the predecessors of this proposal.

Thanks to the authors of "[POP-KEY-DIST]OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution" [POP-KEY-DIST], from where parts of the security considerations where copied.

Thanks to Stefanie Gerdes, Olaf Bergmann, and Carsten Bormann for contributing their work on AS discovery from "Delegated CoAP Authentication and Authorization Framework (DCAF)" [DCAF] (see Section 5.1) and the considerations on multiple access tokens.

Thanks to Jim Schaad and Mike Jones for their comprehensive reviews.

Thanks to Benjamin Kaduk for his input on various questions related to this work.

Thanks to Cigdem Sengul for some very useful review comments.

Thanks to Carsten Bormann for contributing the text for the CoRE Resource Type registry.

Thanks to Roman Danyliw for suggesting Appendix E (including its contents).

Ludwig Seitz and Göran Selander worked on this document as part of the CelticPlus project CyberWI, with funding from Vinnova. Ludwig Seitz has also received further funding for this work by Vinnova in the context of the CelticNext project CRITISEC.

Authors' Addresses

Ludwig Seitz
Combitech
Djäknegatan 31
SE-211 35 Malmö
Sweden
Email: ludwig.seitz@combitech.com

Göran Selander
Ericsson
SE-164 80 Kista
Sweden
Email: goran.selander@ericsson.com

Erik Wahlstroem
Sweden
Email: erik@wahlstromstekniska.se

Samuel Erdtman
Spotify AB
Birger Jarlsgatan 61, 4tr
SE-113 56 Stockholm
Sweden
Email: erdtman@spotify.com

Hannes Tschofenig
Arm Ltd.
6067 Absam
Austria
Email: Hannes.Tschofenig@arm.com