W3C Editor's Draft

Web of Things (WoT) Security Best Practices

W3C Editor's Draft 11 April 2022

▼ **More details about this document**

This version:
https://w3c.github.io/wot-security-best-practices/

Latest published version:
https://www.w3.org/TR/wot-security-best-practices/

Latest editor's draft:
https://w3c.github.io/wot-security-best-practices/

Editors:
Elena Reshetova (Intel Corp.)
Michael McCool (Intel Corp.)

Contributors
In the GitHub repository

Repository
We are on GitHub
File a bug
Contribute

Abstract

This document provides non-normative guidance on how to implement Web of Things (WoT) using best practices for security and privacy. When doing security testing, use of these best practices is assumed.

Status of This Document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

> EDITOR'S NOTE: The W3C WoT WG is asking for feedback
>
> Please contribute to this draft using the GitHub Issue feature of the WoT Security Best Practices repository.

This document was published by the Web of Things Working Group as an Editor's Draft.

Publication as an Editor's Draft does not imply endorsement by W3C and its Members.

This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 2 November 2021 W3C Process Document.

Table of Contents

§ 1. Introduction

For a general discussion of WoT security and privacy issues, see the WoT Security and Privacy Guidelines document.

For details on the Web of Things architecture, please refer to the following:

- the Interest Group web site
- the Working Group web site
- the WoT Architecture document,
- the WoT Thing Description document,
- the WoT Binding Templates document, and
- the WoT Scripting API document.

§ 2. Secure Transport

Secure transport is the foundation of many other security mechanisms, which are vulnerable if it is not used. For example, basic/digest passports and bearer tokens (used in OAuth2) can be intercepted by attackers on the network if transport is not encrypted. Enabling secure transport is essential despite the challenges of using it especially in isolated or local networks.

In general, the recommendation is to use the latest version of TLS and DTLS available, consistent with interoperability requirements. Currently, the latest version of TLS is 1.3 but as this is not yet widely deployed, for interoperability a system may have to be based on TLS 1.2. However, as TLS 1.3 addresses several vulnerabilities in TLS 1.2 in general a migration plan should be in place to TLS 1.3 and new implementations should target TLS 1.3 if possible.

Systems should implement the following for each of the given protocols:

HTTPS:
HTTP + TLS 1.3

CoAPS:
CoAP + DTLS. See also:

- IETF RFC7925: Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things
- IETF RFC7252: The Constrained Application Protocol (CoAP)

MQTTS:
MQTT + TLS 1.3. See also:

- OASIS Message Queuing Telemetry Transport (MQTT) TC
- MQTT Security Subcommittee
- Standard URL scheme for "mqtts://..." (Note: a draft IETF RFC for the MQTT URL scheme is being discussed)

§ 2.1 Global Networks

To do. Use of public URLs, certificates, and CA.

§ 2.2 Offline and Local Networks

We define a local network as one in which not all endpoints are visible to the rest of the internet. This can include both fully offline isolated networks, such as those often used for factory automation, and networks behind a NAT or firewall, as are common for home or business networks. In these cases, establishing the identity of endpoints based on publically visible URLs and the CA system is not possible.

In order to secure HTTP and COAP with TLS/DTLS in local networks, we highly recommend the usage of TLS 1.3 with Raw Public Keys as specified in RFC8446 and RFC7250. However, the keys still need to be assigned and the identities of endpoints established. In local networks, this should be accomplished using an onboarding process, discussed in the next section.

> EDITOR'S NOTE: Local Security Best Practices Under Discussion
>
> Best practices for local security are still under discussion.

Unfortunately it may not be possible in general to use some useful consumers, in particular browsers, with this infrastructure. The best option is to include client devices in the onboarding practice and register a certificate that the browser can use. This is the only option for fully isolated networks. The second best option, usable only for segmented networks with a NAT and/or firewall, is to expose a limited number of secure endpoints, for example a "dashboard" service on a home hub, to the internet, and give it a public URL and certificate. This can be accomplished using either STUN/TURN to set up a tunnel through the NAT and a system such as DynDNS to establish the public IP, or a cloud-based (reverse) proxy. Of course such exposed endpoints should have strong authentication requirements (OAuth2 is recommended). Individual IoT devices in such a system should generally not be directly exposed to the internet.

§ 3. Onboarding

Onboarding is the process of establishing trust with new endpoints in a network, which includes establishing a way to confirm their identities and establishing a mechanism to share encryption keys. This process is essential in order to establish secure transport between endpoints. For systems that are globally visible on the internet, the CA (Certificate Authority) service can be considered an onboarding process that assigns certificates to endpoints. The URL used to access endpoints then becomes the endpoints' identity, and is encrypted into the certificate that is provided. For devices on local networks, keys can be assigned during the onboarding process, but a mechanism to establish the identities of endpoints is also needed.

See the following ITRF/IETF references: Terminology and processes for initial security setup of IoT devices, Different Aspects of Onboarding for IoT/Edge Devices, BRSKI, and SZTP. Some related work in the W3C is also relavant, in particular Decentralized Identifiers (DID) and Verifiable Credentials (VC).

> ISSUE 29: Scripting API Issues Related to Security
>
> The WoT Scripting API needs to establish a secure environment for a script to run in and also needs to expose and consume secure network endpoints. Several of the open issues in the WoT Scripting API depend on the need for secure onboarding, specifically the provision and management of keys and identities.

§ 4. Authentication and Access Control

The best practices for authentication and access control depend on the protocol. In most cases, authentication schemes should only be considered secure when used in combination with secure transport. We recommend the following combinations:

- HTTPS with one of oauth2, bearer, basic, or digest security schemes.
- CoAPS with one of psk, public, or cert security schemes.
- MQTTS with basic AND psk (MQTT native username/password with psk for encrypted communication)

In addition, TDs with HTTP/nosec and CoAP/nosec should be tested and properly handled. They are useful in conjunction with proxies that layer on one of the above secure transport and authentication schemes.

"Local HTTPS" is still a topic of discussion. In addition to the above schemes, using HTTPS with psk, public, or cert schemes to share keys to be used for TLS transport is also acceptable for machine-to-machine communication. However,

currently such schemes may require the user to manually install or accept keys or certificates when using a browser.

§ 4.1 OAuth2 Flows

OAuth 2.0 is an authorization protocol widely known for its usage across several web services. It enables third-party applications to obtain limited access to HTTP services on behalf of the resource owner or of itself. The protocol defines the following actors:

- Client: an application that wants to use a resource owned by the resource owner.
- Authorization Server: An intermediary that authorizes the client for a particular `scope`.
- Resource: a web resource
- Resource Server: the server where the resource is stored
- Resource Owner: the owner of a particular web resource. If it is a human is usually referred to as an end-user. More specifically from the RFC:
  - An entity capable of granting access to a protected resource.

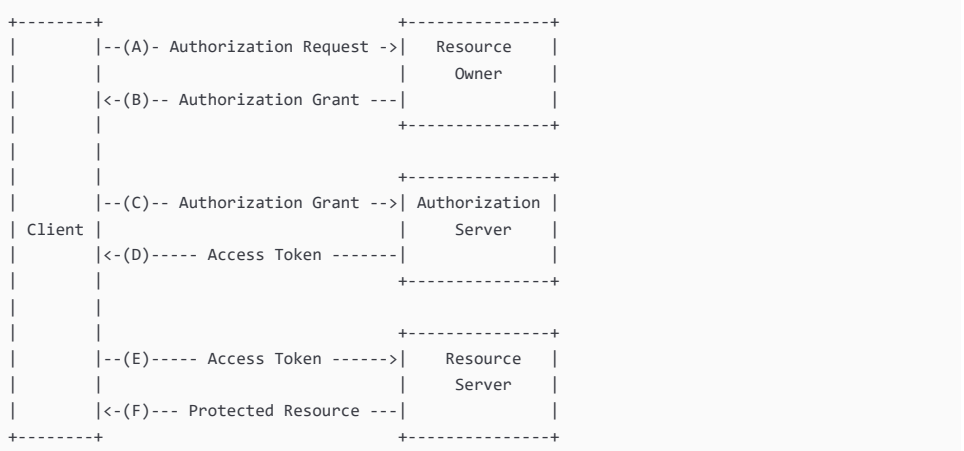
These actors can be mapped to WoT entities:

- Client is a WoT Consumer
- Authorization Server is a third-party service
- Resource is an interaction affordance
- Resource Server is a Thing described by a Thing Description acting as a server.

  May be a device or a service.
- Resource Owner might be different in each use case. A Thing Description may also combine resources from different owners or web server.

> EDITOR'S NOTE
> Editor's note: Check the OAuth 2.0 spec to determine exactly how Resource Owner is defined.
>
> Is it the actual owner of the resource (eg running the web server) or simply someone with the rights to access that resource?

The OAuth 2.0 protocol specifies an authorization layer that separates the client from the resource owner. The basic steps of this protocol are summarized in the following diagram:



Steps A and B defines what is known as authorization grant type or flow. What is important to realize here is that not all of these interactions are meant to take place over a network protocol. In some cases, interaction with with a human through a user interface may be intended. OAuth2.0 defines 4 basic flows plus an extension mechanism. The most common of which are:

- `code`
- `implicit`
- `password` (of resource owner)
- `client` (credentials of the client)

In addition, a particular extension which is of interest to IoT is the `device` flow. Further information about the OAuth 2.0 protocol can be found in IETF RFC6749. In addition to the flows, OAuth 2.0 also supports scopes. Scopes are identifiers which can be attached to tokens. These can be used to limit authorizations to particular roles or actions in an API. Each token carries a set of scopes and these can be checked when an interaction is attempted and access can be denied if the token

does not include a scope required by the interaction. This document describes relevant use cases for each of the OAuth 2.0 authorization flows.

§ 4.1.1 Expected Devices

To support OAuth 2.0, all devices must have the capability of:

- Both the producer and consumer must be able to create and participate in a TLS connection.
- The producer must be able to verify an access (bearer) token (i.e. have sufficient computational power/connectivity).

Comment:

- Investigate whether DTLS can be used.

  Certainly the connection needs to be encrypted; this is required in the OAuth 2.0 specification.
- Investigate whether protocols other than HTTP can be used, e.g. CoAP.
  - found an interesting IETF draft RFC about CoAP support(encrypted using various mechanisms like DTLS or CBOR Object Signing and Encryption): draft-ietf-ace-oauth

§ 4.1.2 Expected Data

Depending on the OAuth 2.0 flow specified, various URLs and elements need to be specified, for example, the location of an authorization token server. OAuth 2.0 is also based on bearer tokens and so needs to include the same data as those, for example, expected encryption suite. Finally, OAuth 2.0 supports scopes so these need to be defined in the security scheme and specified in the form.

§ 4.1.3 Affected WoT deliverables and/or work items

Thing Description, Scripting API, Discovery, and Security.

§ 4.1.4 Description

A general use case for OAuth 2.0 is when a WoT consumer wants to access restricted interaction affordances. In particular, when those affordances have a specific resource owner which may grant some temporary permissions to the consumer. The WoT consumer can either be hosted in a remote device or interact directly with the end-user inside an application.

§ 4.1.5 Variants

For each OAuth 2.0 flow, there is a corresponding use case variant. We also include the experimental "device" flow for consideration.

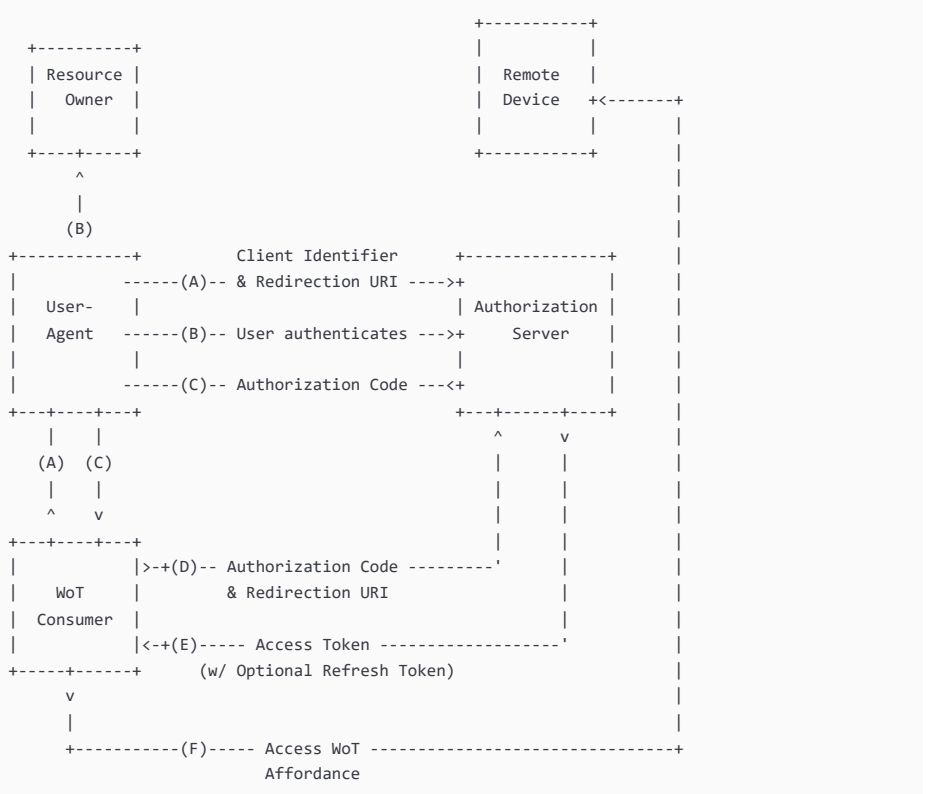
code A natural application of this protocol is when the end-user wants to interact directly with the consumed thing or to grant his authorization to a remote device. In fact from the RFC6749

- Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

This implies that the code flow can be only used when the resource owner interacts directly with the WoT consumer at least once. Typical scenarios are:

- In a home automation context, a device owner uses a third party software to interact with/orchestrate one or more devices
- Similarly, in a smart farm, the device owner might delegate its authorization to third party services.
- In a smart home scenario, Thing Description Directories might be deployed using this authorization mechanism. In particular, the list of the registered TDs might require an explicit read authorization request to the device owner (i.e. an human who has bought the device and installed it).
- ...

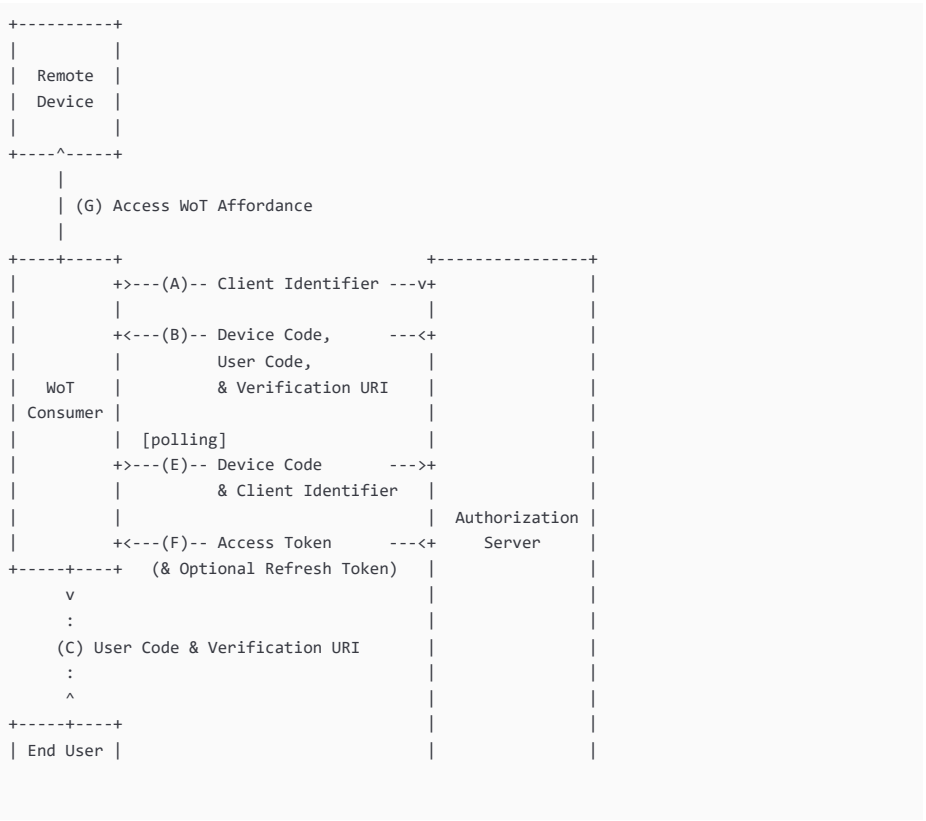
The following diagram shows the steps of the protocol adapted to WoT idioms and entities. In this scenario, the WoT Consumer has read the Thing Description of a Remote Device and want to access one of its WoT Affordances protected with OAuth 2.0 code flow.



Notice that steps (A), (B) and (C) are broken in two parts as they pass through the User-Agent.

device

The device flow (IETF RFC 8628) is a variant of the code flow for browserless and input-constrained devices. Similarly, to its *parent* flow, it requires a close interaction between the resource owner and the WoT consumer. Therefore, the use cases for this flow are the same as the code authorization grant but restricted to all devices that do not have a rich means to interact with the resource owner. However, differently from `code`, RFC 8628 states explicitly that one of the actors of the protocol is an **end-user** interacting with a **browser** (even if section-6.2 briefly describes an authentication using a companion app and BLE), as shown in the following (slightly adapted) diagram:



Notable mentions:

- the protocol is heavily end-user oriented. In fact, the RFC states the following
  - Due to the polling nature of this protocol (as specified in Section 3.4), care is needed to avoid overloading the capacity of the token endpoint. To avoid unneeded requests on the token endpoint, the client *SHOULD* only commence a device authorization request when **prompted by the user and not automatically**, such as when the app starts or when the previous authorization session expires or failAs.
- TLS is required both between WoT Consumer/Authorization Server and between Browser/Authorization Server
- Other user interactions methods may be used but are left out of scope

client credential

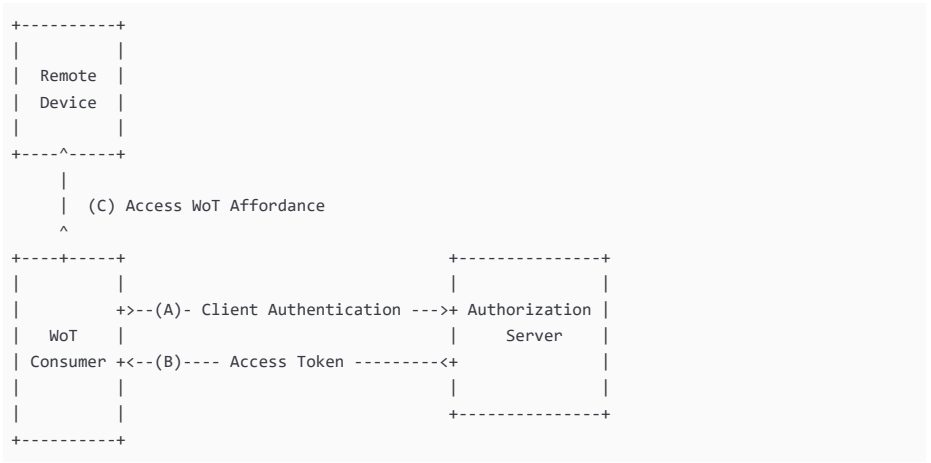
The Client Credentials grant type is used by clients to obtain an access token outside of the context of an end-user.
From RFC6749:

- The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or **those of another resource owner that has been previously arranged with the authorization server** (the method of which is beyond the scope of this specification).

Therefore the client credential grant can be used:

- When the resource owner is a public authority. For example, in a smart city context, the authority provides a web service where to register an application id.
- Companion application
- Industrial IoT. Consider a smart factory where the devices or services are provisioned with client credentials.
- ...

The Client Credentials flow is illustrated in the following diagram. Notice how the Resource Owner is not present.



Comment: Usually client credentials are distributed using an external service which is used by humans to register a particular application. For example, the `npm` cli has a companion dashboard where a developer requests the generation of a token that is then passed to the cli. The token is used to verify the publishing process of `npm` packages in the registry. Further examples are Docker cli and OpenId Connect Client Credentials.

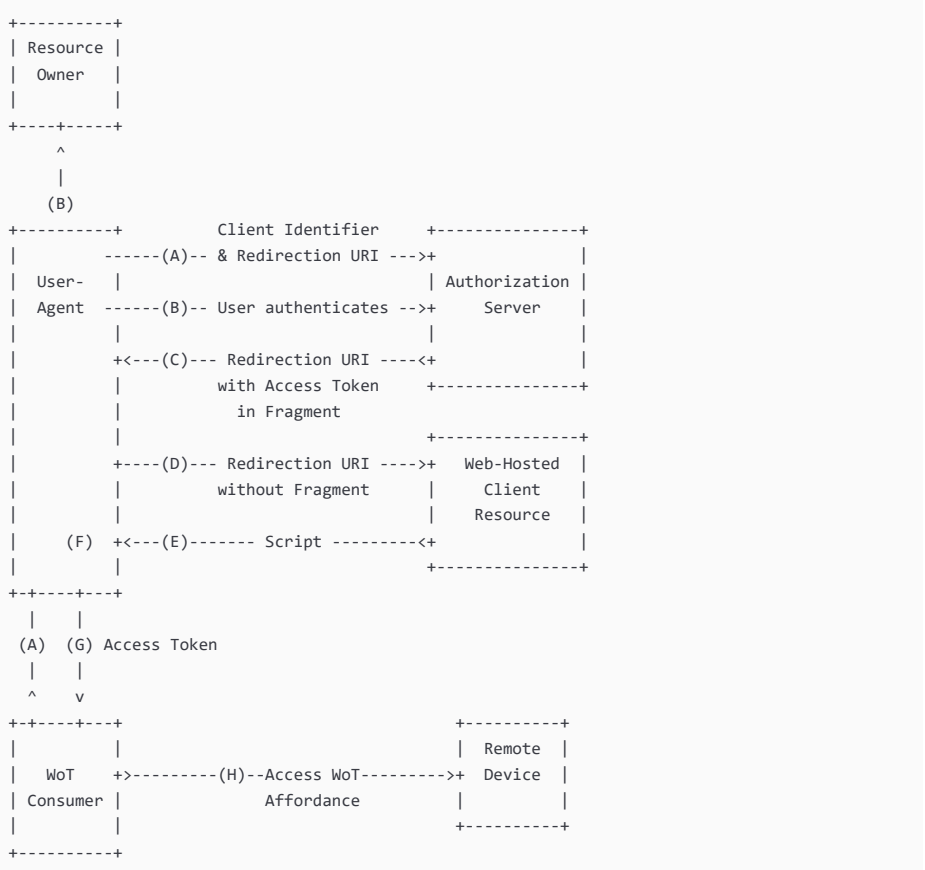
implicit

Deprecated From OAuth 2.0 Security Best Current Practice:

- In order to avoid these issues, clients *SHOULD NOT* use the implicit grant (response type "token") or other response types issuing access tokens in the authorization response, unless access token injection in the authorization, response is prevented and the aforementioned token leakage vectors are mitigated.

The RFC above suggests using `code` flow with Proof Key for Code Exchange (PKCE) instead.
The implicit flow was designed for public clients typically implemented inside a browser (i.e. javascript clients). As the `code` is a redirection-based flow and it requires direct interaction with the resource's owner user-agent. However, it requires one less step to obtain a token as it is returned directly in the authentication request (see the diagram below). Considering the WoT context this flow is not particularly different from `code` grant and it can be used in the same

scenarios.
Comment: even if the `implicit` flow is deprecated existing services may still using it.

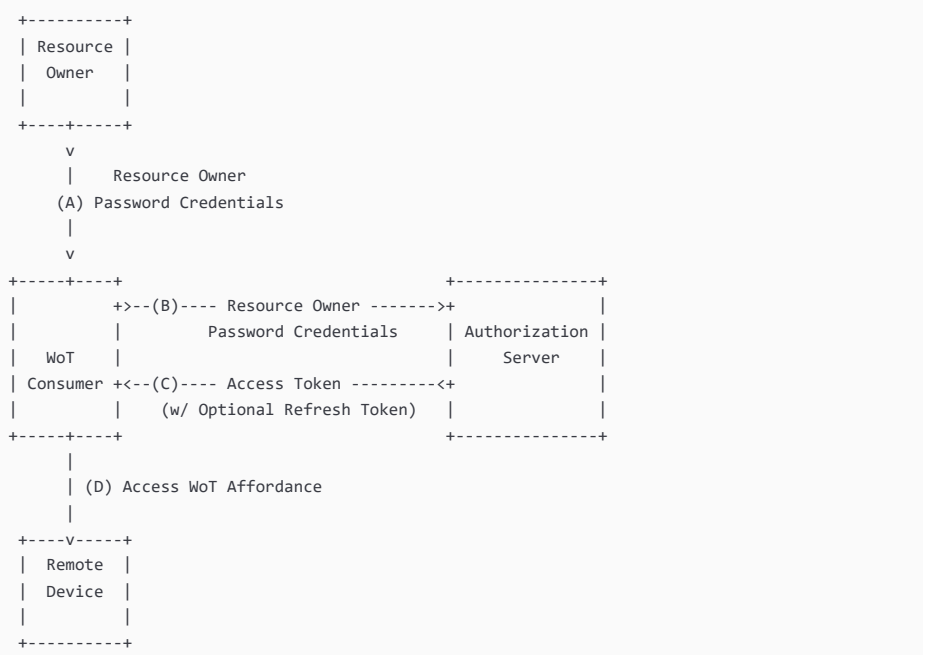


resource owner password

Deprecated From [OAuth 2.0 Security Best Current Practice]:

- The resource owner password credentials grant *MUST NOT* be used. This grant type insecurely exposes the credentials of the resource owner to the client. Even if the client is benign, this results in an increased attack surface (credentials can leak in more places than just the AS) and users are trained to enter their credentials in places other than the AS.

For completeness the diagram flow is reported below.



4.1.6 Security Considerations

See OAuth 2.0 security considerations in RFC6749. See also RFC 8628 section 5 for `device` flow.

4.1.7 Comments

Notice that the OAuth 2.0 protocol is not an authentication protocol, however OpenID defines an authentication layer on top of OAuth 2.0.

5. Thing Directories

Directory services are often used in WoT systems to store TDs and provide discovery services. This is especially useful for devices that need to sleep to conserve battery life. Rather than watching for and responding to discovery requests themselves, they can register their TDs with a directory service which can then respond on their behalf. Directories can either run locally on a gateway (behind a firewall, on the same local network as the devices) or in the cloud (with globally visible URLs). The security considerations and discovery mechanisms are a little different for these two cases. Unfortunately directory services have not (yet) been standardized so our recommendations here are general.

A globally accessible directory service will act much like other web services. It will be available at a "well-known" URL that will have to be configured by the user. Registration of devices will have to be associated with a particular user and use of the service will have to be protected by authentication and confidentiality mechanisms, such as HTTPS combined with one of the authentication mechanisms listed above. The directory service should avoid doing any processing for unauthenticated connection attempts in order to protect itself from DoS attacks.

Local directory services may also offer a web interface but may also advertise their availability using mDNS/Zeroconf. Authentication and confidentiality for a local service can also be secured via HTTPS although the issues of "local HTTPS" also arise for such services; in general, the user may have to "on-board" devices using some kind of pairing approach. If the local service is located on a network located behind a firewall it is possible to depend on link-layer encryption such as WPA2 although this is not as secure as transport-layer security using TLS.

Registering a device's TD with a directory service is also a suitable time to capture user consent for the distribution of the TD and the data from the device. Such consent should include appropriate limits on who can access the data and for how long it can be retained. Also, since personally-identifiable information can be inferred from TDs, TDs should themselves be treated as personally-identifiable information and suitably protected. This means that directories should generally only provide TDs via mutually-authenticated channels to users that are authorized to access those TDs.

6. Object Security

> EDITOR'S NOTE: Object Security Under Construction
>
> We need additional implementation experience to refine this, and also need to consider how the recently proposed TD Signatures align. Some additional TD examples and/or features may be needed. So this section should be considered "under construction".

Object security is recommended if a CoAP or MQTT to HTTP gateway is used that translates protocols. Ideally however you would NOT translate the payload itself but use end-to-end security. It is also important to still use object security with TLS and DTLS; object security alone is generally insufficient. The main advantage of object security is that a compromised Gateway will be prevented from modifying payloads.

Example object-security standards to consider are COSE [RFC8152], and OSCORE [RFC8613].

7. Secure Update and Post Manufacturing Provisioning

The WoT is primarily concerned with the operational phase of devices. It is assumed that devices and other components of a WoT system (gateways, for example) start the operational phase in a secure state. WoT best practices are focused on keeping devices and services in a secure state staring from this assumption. However, to enter operational state in secure fashion, additional best practices need to be followed during manufacturing, deployment and provisioning, and best practices should also be followed for secure update.

Good references for best practices for secure update and provisioning are the IIC Security Framework [IicSF16] and the IoT Security Foundation's guidelines [ISF17].

§ 8. Terminology

Please refer to the WoT Architecture document for terminology definitions.

§ A. Acknowledgements

Cristiano Aguzzi contributed the section on OAuth2 and provided best-practice recommendations. Elena Reshetova contributed to the discussion around lifecycle and transport. Philipp-Alexander Blum and Oliver Pfaff contributed to transport and object security. Ben Francis contributed use case input to local transport.

§ B. References

§ B.1 Informative references

[IicSF16]
The Industrial Internet of Things Volume G4: Security Framework. Industrial Internet Consortium. IIC:PUB:G4:V1.0:PB:20160926. Sept 2016. URL: https://www.iiconsortium.org/IISF.htm

[ISF17]
IoT Security Foundation Best Practice Guidelines. IoT Security Foundation. May 2017. URL: https://iotsecurityfoundation.org/best-practice-guidelines/

[RFC8152]
CBOR Object Signing and Encryption (COSE). J. Schaad. IETF. July 2017. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8152

[RFC8613]
Object Security for Constrained RESTful Environments (OSCORE). G. Selander; J. Mattsson; F. Palombini; L. Seitz. IETF. July 2019. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8613

↥