Delegated Device Authorization in the Web of Things

Jan Romann, University of Bremen

WoT Community Group Meetup, February 23, 2023

About me

- Originally a political science graduate
- Currently pursuing a Master's degree in Computer Science at the University of Bremen, Germany
- Dealing with the Web of Things for about 2 ½ years
    - Focus on CoAP and related technologies as well as Thing Model conversions
- Invited Expert to the WG since September 2021

Universität Bremen

About our Project (NAMIB)

- Bachelor's and Master's project at the University of Bremen
- 19 participants in the Bachelor's phase, 11 participants in the Master's phase
- Lasted about two years (both phases combined), until September 2022
- Supervised by Carsten Bormann, Ute Bormann, and Olaf Bergmann
- Goals: Improving security and interoperability in the IoT

Motivation and Goals

Motivation

- Hotel scenario
  - Guests check in, want to access IoT devices in their rooms
  - Hotel wants to limit permissions by
    - **space** (room-wise)
    - **time** (duration of the stay),
    - **scope/status** (e.g., guests vs staff)
  - Solutions should be interoperable to enable seamless interactions

Main Working Areas (simplified)

Authorized Device Interactions

- ACE-OAuth, CoAP, and WoT
- Core of the hotel scenario
- Main focus of today's talk

Onboarding Solutions

- EAP-{NOOB, UTE}

Restricted Network Access and Intruder Detection

- Manufacturer Usage Description (MUD, RFC 8520)

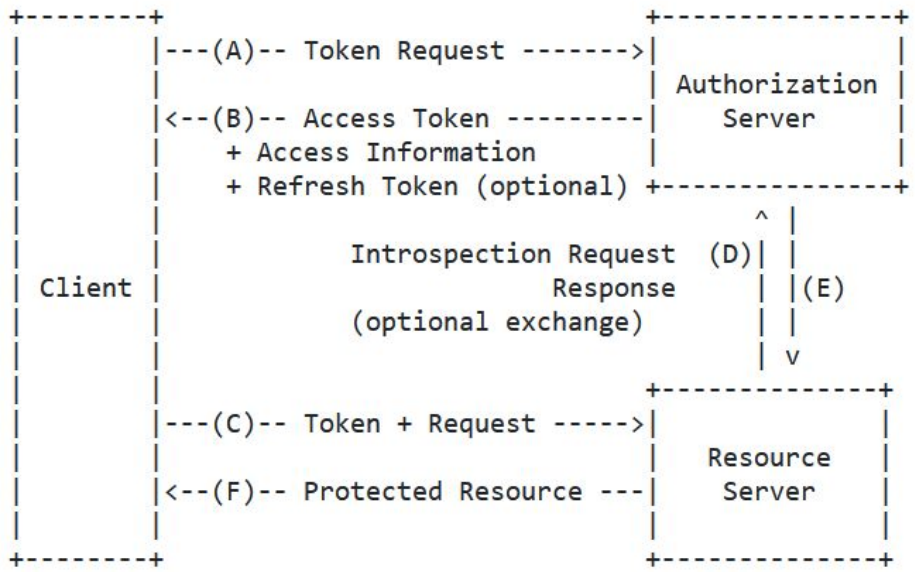
Goals

- Interoperable solution for authorized access to IoT devices
  - → Open standards and implementations
- Investigation of potential specification improvements
  - E.g., better integration of ACE-OAuth and CoAP into the Web of Things

ACE-OAuth (RFC 9200)

- Adaption of OAuth 2.0 for constrained environments
- Delegation of Authorization to an Authorization Server (AS)
- Authorization is commonly granted via credentials (Proof-of-Possession tokens)
  - PSK/RPK credentials for DTLS connection (DTLS Profile, RFC 9202)
  - OSCORE keying material (OSCORE Profile, RFC 9203)
- Access Tokens are (usually) CBOR-encoded

ACE-OAuth (RFC 9200) Protocol Flow



Constrained Application Protocol (CoAP, RFC 7252)

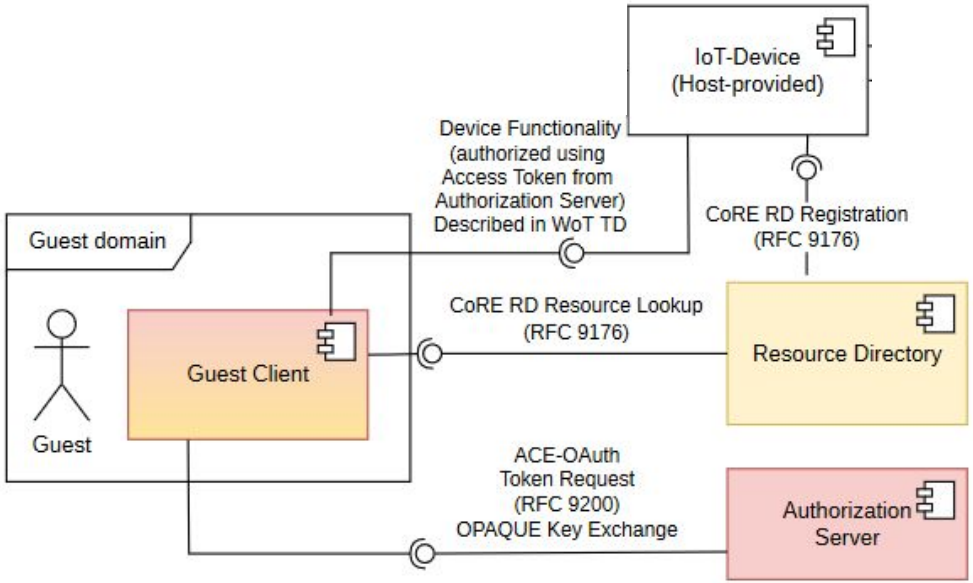
- Web transfer protocol for constrained nodes and environments
  - Primarily UDP-based, with small overhead
- Compatible with HTTP (enables proxying)
- Supports observing resources (RFC 7641) and multicast
- Transport (DTLS) and object security (OSCORE, RFC 8613)
- Variants for TCP and WebSockets exist (RFC 8323)

OPAQUE (draft-irtf-cfrg-opaque)

- Protocol for “asymmetric password-authenticated key exchange” (aPAKE)
- Used to register and log in our guests
  - Generates a shared secret between the guest’s client and the authorization server
- Server never learns the password that is used

Architecture

Architecture (excerpt)



Implementation

Components

- Authorization Server (with check-in mechanism)
- IoT Devices
- Guest Client
- (CoRE Resource Directory for Discovery → aiocoap)

Two Kinds of Libraries

Rust-based libraries (AS, IoT devices)

- dcaf-rs (for ACE-OAuth)
- *libcoap-rs*
- *tinydtls-sys*
- (opaque-ke)

Dart-based libraries (Guest Client)

- dcaf (for ACE-OAuth)
- dart_wot
- *dart_tinydtls*
- *dtls2* (OpenSSL-based)
- (coap)
- (*opaque-dart*)

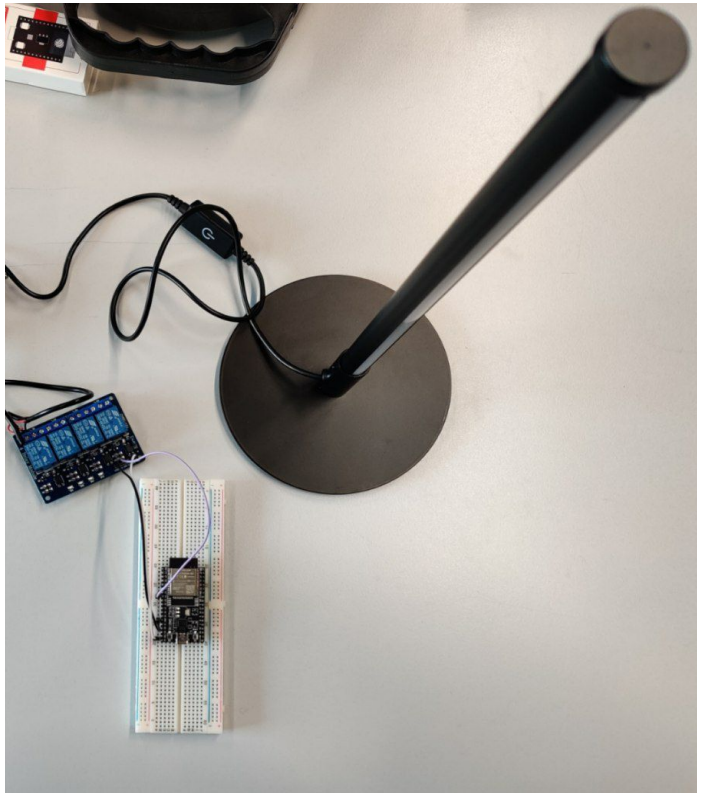
Third party libraries in parentheses. Libraries using FFI bindings in italics.

Authorization Server

- Rust-based implementation based on dcaf_rs and libcoap-rs
- Supports check-in using OPAQUE
- Display of registration data in QR code
  - Includes username, password, discovery URL
- Web interface is still WIP
  - Temporary solution using CLI

IoT Devices

- Implemented on ESP32s using Rust bindings to ESP-IDF
  - Rust support for RIOT was limited at the time
- Re-use of dcaf_rs for ACE-OAuth and wot-td for TD generation
- Drawback: Currently requires an allocator
  - However: Use of ESP32s allowed us to use Rust's standard library



One of our IoT devices: An ESP32 controlling a lamp via a relay.

ACE-OAuth: Security Scheme vs Creation Hint

```
"securityDefinitions": {
    "ace_sc": {
      "scheme": "ace:ACESecurityScheme",
      "ace:as": "coaps://as.example.com/token",
      "ace:audience": "coaps://rs.example.com",
      "ace:scopes": ["rTempC"],
      "ace:cnonce": true
    }
  },
  "security": ["ace_sc"],
```

```
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'
}
```

ACE-OAuth: Different Scopes, Same Resource

```
"temperature": {
    "minimum": 15,
    "maximum": 24,
    "forms": [
        {
         "href": "/temperature",
         "ace:scopes": ["guest", "staff"]
        }
    ]
},
```

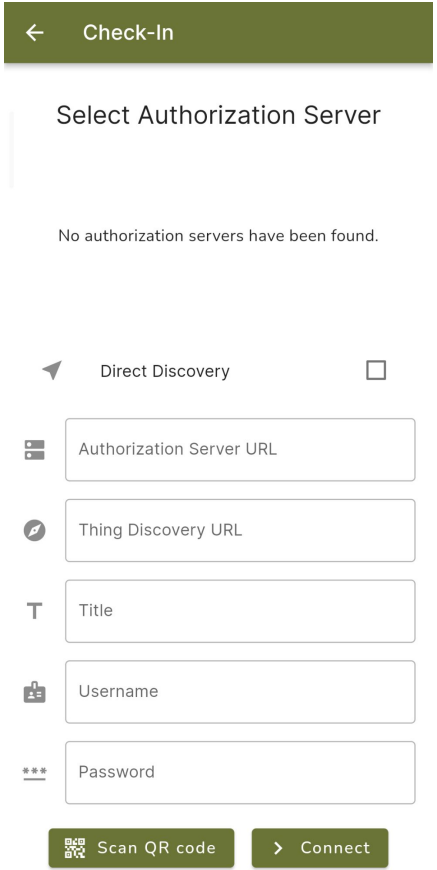
```
"temperatureStaff": {
    "forms": [
        {
         "href": "/temperature",
         "ace:scopes": ["staff"]
        }
    ]
},
```

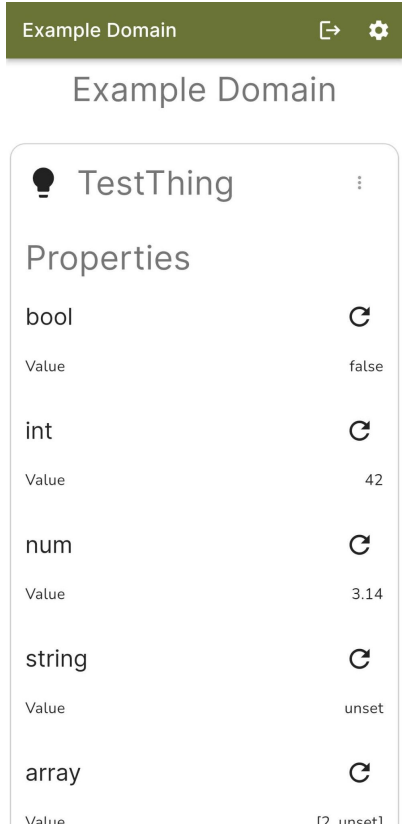
Guest Client

- For CoAP, we needed UDP sockets
  - → Using a web client was not possible
- Re-using node-wot with React Native seemed difficult to achieve
  - Not all node packages are compatible (especially node-coap)
- Native Android would have been an option
  - But: This would have meant no potential Cross-Platform support

Guest Client

- Solution: Using Flutter with Dart
- Potential Support for five native platforms
  - Android
  - iOS
  - Linux
  - macOS
  - Windows
- Benefits: Strong type system and compilation to native code



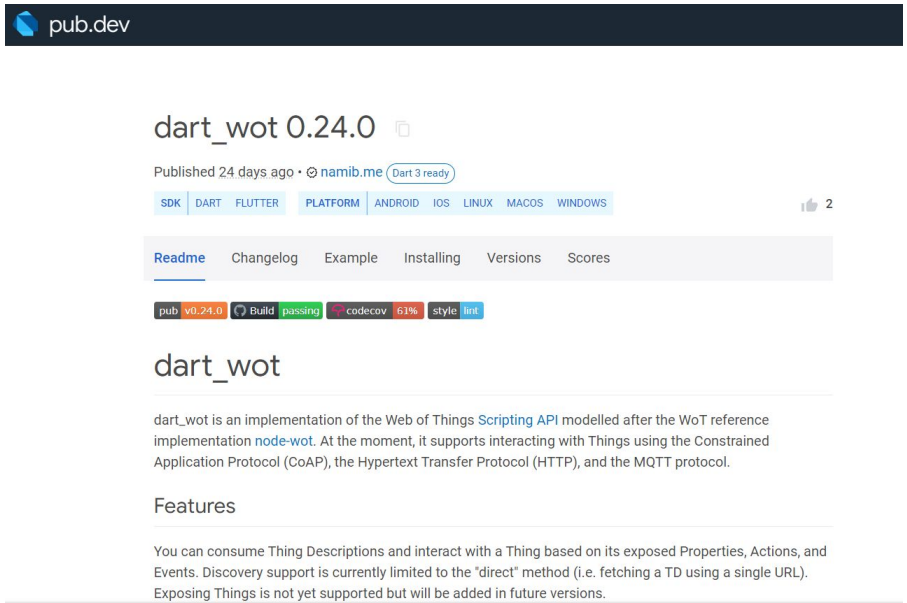


Guest Client

- However:
  - There was no WoT and no ACE-OAuth implementation in the Dart ecosystem yet
  - The available CoAP library for Dart did not have DTLS support
- → Creation of new libraries and contributions to existing ones (especially coap)

WoT-Implementation for Dart: dart_wot

- Modelled after node-wot
- Support for HTTP, CoAP, and MQTT
- So far only Consumer and Discoverer
- Support for Security Bootstrapping and various Discovery methods
- Builds upon dcaf and the (DTLS-enchanced) coap library



Discovery (direct)

```dart
Future<void> main() async {
  final servient = Servient()..addClientFactory(CoapClientFactory());

  final wot = await servient.start();
  final uri = Uri.parse('coap://plugfest.thingweb.io/testthing');

  // .discover() returns a Stream (that can also be used like an AsyncIterator)
  await for (final thingDescription in wot.discover(uri)) {
    await handleThingDescription(wot, thingDescription);
  }
 }
```

Discovery (CoRE Link Format + Multicast)

```dart
Future<void> main() async {
  final servient = Servient()..addClientFactory(CoapClientFactory());

  final wot = await servient.start();
  final uri = Uri.parse('coap://[ff02::1]/.well-known/core');

  await for (final thingDescription in wot.discover(uri, method: DiscoveryMethod.coreLinkFormat)) {
    await handleThingDescription(wot, thingDescription);
  }
}
```

ClientSecurityProvider

```
final clientSecurityProvider = ClientSecurityProvider(
  aceCredentialsCallback: (uri, form, creationHint, invalidCredentials) async => ...,
  pskCredentialsCallback: (uri, form, identityHint) => ...,
  basicCredentialsCallback: (uri, form, invalidCredentials) async => ...,
  ...
);
```

Credentials and Security Bootstrapping

```dart
final clientSecurityProvider = ClientSecurityProvider(
  aceCredentialsCallback: (uri, form, creationHint, invalidCredentials) async =>
      retrieveAceCredentials(uri, form, creationHint, invalidCredentials),
);

Future<void> main() async {
  final servient = Servient(clientSecurityProvider: clientSecurityProvider)
    ..addClientFactory(CoapClientFactory());

  final wot = await servient.start();
  final uri = Uri.parse('coap://[fe80::db8:abcd]/.well-known/wot');

  await for (final thingDescription in wot.discover(uri)) {
    await handleThingDescription(wot, thingDescription);
  }
 }
```

Limitations and Problems

Limitations

- Implementation for IoT-Devices not yet suitable for constrained devices
  - Need to be adapted to no_std environments
  - (De)serialization of Thing Descriptions is too costly
- Missing library features (e.g., OSCORE support)
- OPAQUE works for ACE-OAuth but needs to be explicitly supported
  - Could be specified as mechanism for trust establishment (draft currently WIP)
- Actual applications are not published yet

Problem: Binding to Native Libraries

- In order to close existing implementation gaps, we decided to bind to pre-existing projects written in C using FFI bindings from both Rust and Dart
- This lead to…
  - libcoap-rs and tinydtls-sys for using CoAP with the Authorization Server
  - dart_tinydtls and dtls2 for adding DTLS support to the Dart CoAP library
  - Use of the pre-existing opaque-dart as a binding to libopaque (unfortunately with limited macOS and iOS support)

FFI is great! But…

- Cross-platform was very difficult to achieve with dart_tinydtls
  - Building for Windows failed with newer versions of ffi package
  - macOS and iOS have a strict policy for binaries and a limited build system when using Flutter
    - Caused problems for using tinydtls (and also opaque-dart) on these platforms
  - → New dtls2 package reusing pre-existing code and OpenSSL binding
- libcoap has a complex memory model
  - Creation of Rust wrapper was challenging

Conclusion

Conclusion

- Architecture and implementation already worked
- ACE-OAuth can be used for delegated Authorization in the Web of Things
- However: More specification and implementation work is needed

Future Work

- Specification of OPAQUE as an onboarding mechanism for ACE-OAuth?
- New TD Security Schemes for ACE-OAuth et al.
- More compact self-descriptions for constrained devices
  - code size of (de)serialization logic is too large (at least 20 KiB)
- Guidelines/components for generating UIs based on TDs?
- Wider “native” support for DTLS in different ecosystems is desperately needed

Thank you for your attention! :)

Find us on GitHub

https://github.com/namib-project