Towards multi-party policy-based access control in federations of cloud and edge microservices

Davy Preuveneers and Wouter Joosen
imec-DistriNet, KU Leuven
Department of Computer Science
Heverlee, Belgium
{firstname.lastname}@cs.kuleuven.be

***Abstract*—The development and deployment of microservices and containers come with a promise of flexibility by embracing heterogeneity and reducing the amount of communication and coordination between service teams. However, when such software ecosystems are developed in large organizations with a high degree of independence, and deployed in the cloud and at the edge, security becomes a non-trivial concern. The challenge that we address in this work is the delegated management of access control decisions to multiple stakeholders in continuously evolving federations of cloud and edge microservices. To ensure that user-centric access control remains sustainable in such complex service delivery models, we present a dynamic granular access control solution on top of different authorization frameworks. By leveraging microservice technologies, our solution is flexible, scalable, and contextual, and can adhere to the security needs of different stakeholders in microservice federations – from DevOps teams to common end-users – with the necessary agility to respond to exceptional security circumstances.**

***Index Terms*—authorization; microservices; policy-based access control; edge; federation**

I. Introduction

Microservices [1] offer a convincing competitive advantage for building sophisticated distributed applications as a composition of services that each implement a specific functionality. As advocated by James Lewis and Martin Fowler [2] and as depicted in Fig. 1, the guiding principles behind microservices are that (1) a microservice only offers a limited and clearly defined set of functions, (2) a microservice can be developed independently using the most appropriate programming language, (3) each microservice is stateless and manages its own database, and (4) each microservice runs autonomously in its own process. The low coupling and high cohesion of such distributed choreographies of independent microservices simplify the introduction of new functionalities, and they allow for each microservice to be developed, deployed, modified and scaled out independently.

As the adoption of microservices in the enterprise and their deployment in the cloud and at the egde grow, security and privacy become first class concerns, and having a solid access control solution for microservice APIs and the content they expose is key [3]. In practice, API gateways and service proxies are commonly used as API intermediaries that offer an elegant location to enforce microservice API access control policies. However, when multiple parties or stakeholders are involved [4], [5] in the evaluation of access control decisions,

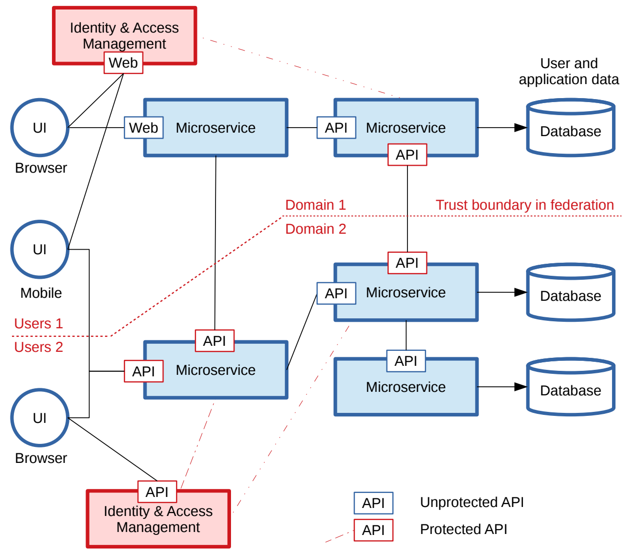

Fig. 1. High-level decomposition of microservice based API consumers and providers in a federation of two collaborating enterprises, each with their own users and access control solutions

such as in mixed cloud, fog and edge deployments, implementing the security measures becomes much more complicated:

- Access control to content and services is in part delegated to end-users and non-security experts.
- Collaborating organizations independently enforce their own access control decisions but do not necessarily share the sames rules, roles and other subject attributes.

For each of the above scenarios, standardized and state-of-practice token (OAuth 2.0 [6] and UMA 2.0 [7] using signed JSON Web Tokens) and policy-based (XACML 3.0 [8], NGAC [9], OPA [10], etc.) authorization frameworks exists to effectively handle security permissions and entitlements. However, when they all must be enforced in a unified microservice offering, delegation of access control and granting permissions must be addressed in a holistic manner:

- *Person-to-self*: a third party application accessing resources on a user's behalf (cfr. OAuth 2.0).
- *Person-to-person*: pro-actively sharing resource in a selective manner to other persons (cfr. UMA 2.0).

DOI 10.1109/EuroSPW.2019.00010

- *Person-to-organization*: grant non-individuals access to resources (cfr. UMA 2.0).

The managerial complexity of state-of-practice security measures often impedes swiftly setting up federations or mesh networks of microservices spanning multiple operational domains. To ensure that access control remains sustainable in such sophisticated service delivery models, we present a dynamic granular access control solution that is flexible, scalable, and contextual. By leveraging microservice technologies, our solution can adhere to the security needs of different stakeholders in microservice federations − DevOps teams, service providers and consumers, content owners and other end-users − with the necessary agility to respond to exceptional security circumstances and emerging security threats.

The key contributions of this research extend open source building blocks and can be summarized as follows:

1) Delegated authorization to multiple stakeholders (service providers and consumers, content owners, etc.)
2) Independent deployability and manageability of authorization policies and supporting runtime frameworks
3) Service orientation of policy-based access control supporting interoperability across domains in a federation

This work is carried out in the frame of the FUSE research project[1] which aims to provide a microservice-oriented, container-based service environment to deploy and manage software on federated domains with the ability to quickly set up ad hoc federations and secure communication between domains with minimal intervention.

The remainder of this paper is structured as follows. In section II we review relevant related works on authorization and access control. Our multi-party access control solution and its building blocks are presented in section III. In section IV, we discuss the experimental results with our framework from a performance and scalability perspective, and compare the outcome with the related work. Finally, section V summarizes the main results and gives concluding remarks.

II. Related work

This section reviews several state-of-practice and state-of-the-art solutions for access control in service-oriented software platforms.

The OAuth 2.0 authorization framework [11] in tandem with JSON Web Tokens (JWT) [12] have become the industry standard in providing secure access to web APIs. The advantage of JWT tokens is that these tokens are designed to self-contain all the information about subjects, what the subjects need, and what they are authorized to do. They can be transmitted between parties, verified and trusted because they are digitally signed. These properties make the bearer tokens a good fit for stateless environments. However, the scope based authorization of such tokes is often coarse grained and less flexible to define context-dependent conditional access patterns.

[1]https://www.imec-int.com/en/what-we-offer/research-portfolio/fuse

The User-Managed Access (UMA) 2.0 specification [13] is an OAuth-based access management protocol standard for selective party-to-party sharing, designed for individuals to authorize who and what can get access to their data, content, and services. The asynchronous nature of the authorization protocol enables requesting, granting and revoking access between consumers and resource owners in a user-friendly manner.

Policy-based access control models are a state-of-practice solution to grant permissions based on a policy defined external to the service implementation. Contrary to token ownership-based approaches, such models use digital policies, comprised of logical rules, to guide authorization decisions for accessing online services and resources. In Attribute Based Access Control (ABAC) [9] [14], the authorization decisions in such policies are made based on a set of characteristics, or attributes, associated with the requester, the environment and/or the resource itself. While eXtensible Access Control Markup Language (XACML) [9] is a well-known and state-of-practice attribute-based access control language, the XML nature of the language is rather verbose and inefficient to parse, which is why JSON-based languages are being proposed [15] [16] that are more simple to process but equally expressive. The Open Policy Agent (OPA) [10], hosted by the Cloud Native Computing Foundation (CNCF), is a general-purpose policy engine for fine-grained control across cloud native environments, including admission control in Kubernetes. The policies are written in the Rego language supporting declarative rules for processing JSON-based data documents.

Squicciarini et al. [4] explored limitations in contemporary access control solutions, and argue that most of them are single-user or single-owner oriented with a focus on confidentiality rather than facilitating controlled sharing. Access decisions are often binary, or based on inflexible policies. Indeed, overriding security policies in a fine-grained and controlled manner but only under exceptional circumstances is not trivial. In healthcare, this need to override in emergency circumstances is known as the break-the-glass [17] bypass procedure. However, the bypass is very coarse grained, i.e. access is granted and post-factum the healthcare professionals must motivate why they applied the procedure, possibly followed up by an external audit of the access log.

It is this gap in the state-of-the-art on access control that this work aims to bridge by leveraging the simplicity of tokens and the flexibility of policies. Our goal is a multi-party access control solution where end-users or stakeholders from different operational domains can jointly own and grant access to content and resources such that fine-grained authorization can be managed both at the individual microservice and at the federation level.

III. Multi-party policy-based access control

Before we discuss the technical building blocks of our access control solution, we will first introduce a simple home security motivating scenario involving microservices belonging to different domains. The scenario serves as a proof-

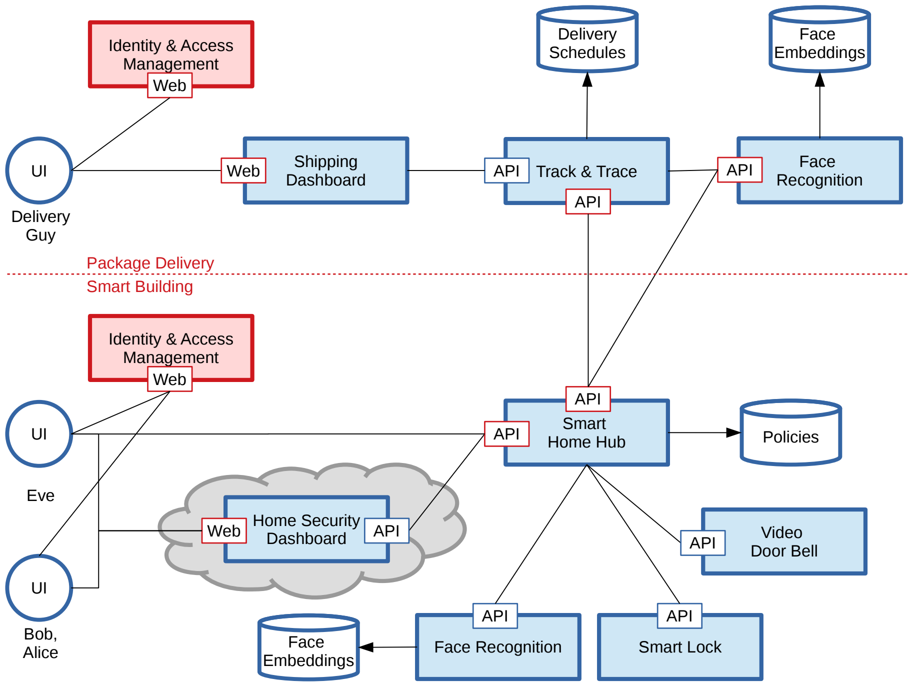

Fig. 2. High-level decomposition of microservices in the cloud and at the edge for the in-home delivery use case

of-concept application to illustrate how different end-users and stakeholders from different domains can play a role in authorization decisions.

A. Motivating use case: in-home delivery with smartlocks

Smartlocks are aimed at ensuring a deadbolt can automatically lock and unlock. These smartlocks are usually battery powered, and connect to a smartphone over Bluetooth, or to the internet over a WiFi connection via a Bluetooth-enabled base station. When the smartphone is not in the direct vicinity of the smartlock, remotely unlocking the deadbolt over a direct Bluetooth or WiFi connection will not be possible. Remote smartlocks therefore rely on a cloud SaaS application that acts as an intermediary between the end-user's smartphone and the smartlock to manage and monitor the home security system remotely. The SaaS application can also assist with transferring the camera footage of the video doorbell to capture who is at the door. This way, owners can unlock the door to have their packages securely delivered inside the home.

Various such custom intelligent deadbolt access control solutions have popped up on the market over the past few years, with the Amazon Key Smart Lock Kit[2], being a noteworthy example. The following user story also addresses in-home delivery for an apartment building where the main entrance door of the building has a video door bell and a smart lock.

The fact that security and access control is not always adequately addressed, was demonstrated for the Amazon Ring Doorbell application with the CVE-2019-9483[3] vulnerability, highlighting how bad encryption allows attackers to obtain audio and video data, or inject counterfeit video traffic that does not correspond to the actual person at the door, hereby effectively compromising home security.

- Bob and Alice and their son Charlie live in an apartment on the second floor of a 3-story building. They can remotely manage the smart lock.
- Bob expects a package and wants to temporary override the default access control policies to grant access to the delivery guy.
- Bob can monitor the package delivery and the location of the delivery guy between the expected delivery times or when in close proximity.
- Eve lives in the apartment on the first floor. She can also remotely unlock the main entrance door.

Fig. 2 depicts the simplified federation of microservices with various stakeholders having different roles and security needs. The physical smart lock at the main entrance door of the building is managed by the *Smart Lock* microservice. The *Smart Home Hub* integrates the smart lock with the camera footage of the *Video Door Bell* microservice. When a direct connection with the hub service is not possible, the *Home Security Dashboard* SaaS application in the cloud acts as an intermediary. Each of the *Face Recognition* microservice instances recognize different individuals within a picture. Our proof-of-concept implementation is based on the FaceNet [18] deep neural network. The *Track & Trace* microservice offers details about package deliveries, and the current location of the delivery guy. For authentication, we rely on the OpenID

[2]https://www.amazon.com/keyinhome
[3]https://nvd.nist.gov/vuln/detail/CVE-2019-9483

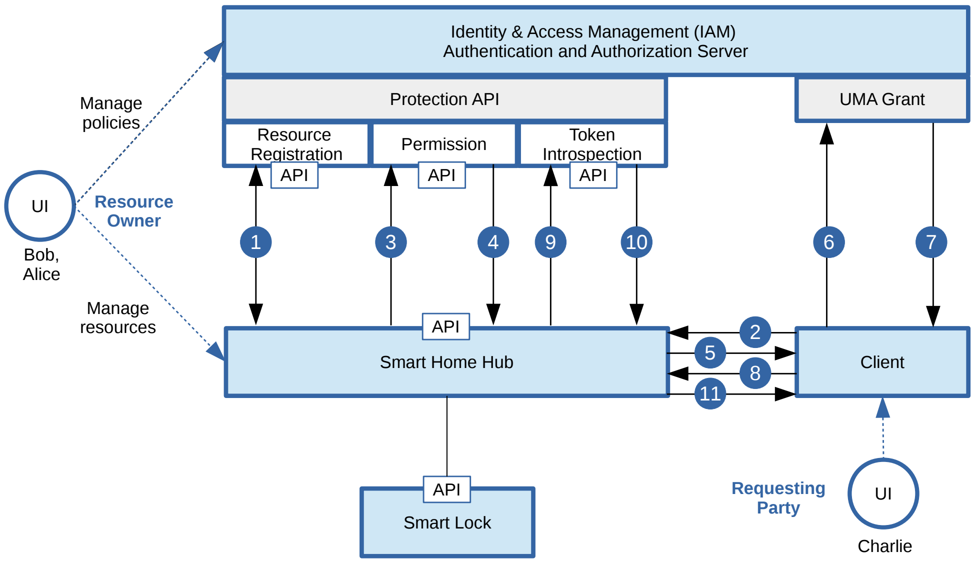

Fig. 3. User-Managed Access (UMA) 2.0 federated authorization to manage access to the smart lock resource

Connect protocol [19], which is supported by many off-the-shelf Identity and Access Management (IAM) solutions.

B. *Use case analysis and security requirements*

The aim is to analyze to what extend the access control for the above in-home delivery scenario can be realized on top of state-of-practice authorization solutions. We will therefore elicit the most important security requirements for which state-of-practice solutions fail to fit a good solution.

1) **Multiple resource owners**: The smart lock resource is jointly owned and managed by multiple parties (i.e. Bob, Alice, Eve and the other neighbors).
2) **Convenient consent**: Charlie's parents can grant him consent to open the smart lock, but deny any administrative privileges. Consent can be easily revoked.
3) **Delegated authorization**: Bob and Alice should be able to delegate authorization decisions to one another, but Eve should not be able to override the authorization policy decisions by Bob and Alice.
4) **Multi-party authorization**: The delivery guy can only enter the building when authorized by the package delivery company and either Bob or Alice.
5) **Usage control**: Access to the track & trace microservice is conditionally granted not only to handle authorization requirements but also to meet performance SLAs.
6) **Selective consent**: Bob is granted access to the location of the delivery guy, but only between certain time slots or when within a certain proximity of the destination.
7) **Selective policy overriding**: The default policy of Bob and Alice needs to be temporarily overruled to grant the delivery guy access to the building.
8) **Privacy and minimal information disclosure**: Only the delivery guy should be authorized by the company (e.g. after face recognition and delivery schedule check), not every visitor of Bob and Alice.

Obviously, all control and data flow interactions between users and microservices as well as among microservices should be encrypted to guarantee integrity, confidentiality and origin authenticity.

Based on the above set of challenges, it is clear that a simple token based authorization approach – be it OAuth 2.0 or UMA 2.0 – will not suffice to handle the more complex authorization policies. Furthermore, sharing and delegating access control across multiple resource owners is not trivial either. In the following subsection, we will elaborate on our solution to address these concerns.

C. *Policy-based access control to microservice APIs*

Each party in the federation has its own identity and access management platform through which individuals, i.e. the different service consumers, are authenticated by means of the OpenID Connect protocol. The authorization flow follows the User-Managed Access (UMA) 2.0 specification – i.e. the UMA 2.0 Grant for OAuth 2.0 Authorization[4] and the Federated Authorization for UMA 2.0[5] – to grant access to the *Smart Lock* microservice is illustrated in Fig. 3. The actual access control rules and conditions under which authorization is granted and how the rules are declared in a policy language like XACML or OPA are beyond the scope of the UMA specification, as policy expression and evaluation are done out of band. The *Smart Home Hub* microservice acts both as an edge API gateway to the smart lock and as the *resource server*. These are the steps:

[4]https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html
[5]https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-federated-authz-2.0.html

1) Bob or Alice – being the *resource owners* – register the Smart Lock resource at the *Resource Registration Endpoint* of the *authorization server*, and configure who can access which resources under which permission scopes.
2) The mobile *client* of Charlie – the son of Bob and Alice and the *requesting party* – makes an access request to the protected smart lock resource, but without a valid *Requesting Party Token* (RPT).
3) The resource server requests the permissions (i.e. the *scopes*) bounded to smart lock resource from the *Permission Endpoint* of the authorization server.
4) The *Permission Endpoint* returns a permission ticket.
5) The resource server returns the URI of the authorization server and the permission ticket to Charlie's client.
6) The client submits the permission ticket as well as any necessary claims to the UMA Grant endpoint to retrieve an access token.
7) The UMA Grant endpoint of the authorization server issues an RPT token to Charlie's client.
8) The client again requests access to the protected smart lock resource, but now with a valid RPT token.
9) The resource server introspects the RPT token at the *Token Introspection Endpoint* of the authorization server.
10) The authorization server returns the status of the token introspection.
11) The resource server grants access to the protected smart lock resource.

The above person-to-person sharing scenario assumes that Charlie can authenticate against the same authorization server that protects the resources so that relevant identity claims can be gathered (e.g. with the OpenID Connect protocol) for evaluation of the access control policies at the authorization server. Such an access control policy could check whether the last name and home address of the subject correspond to either those of Bob and/or Alice.

D. Access control decisions with multiple resource owners

Note that next to Bob and Alice, Eve is also considered a resource owner of the smart lock microservice. When Charlie aims to request access, only the policies bounded to the smart lock resource and the scope defined by Bob and Alice should be considered when evaluating decisions to grant access.

To support multiple owners of a resource, and isolate per owner the evaluation and enforcement of the access control policies, the requesting party must be able to indicate the relevant resource owner in step 2 and 3 when requesting an RPT. When access is denied and the requesting party submits a request to demand access, the authorization server needs to know which resource owner should provide consent. Furthermore, in the above scenario, if consent to the smart lock resource is granted by Bob, we may want to let Alice – but not Eve – be able to revoke that consent. One pragmatic way of handling this case is by passing the necessary claims about ownership as a custom parameter or header in the HTTP request in step 2, and limit the applicability of permissions and

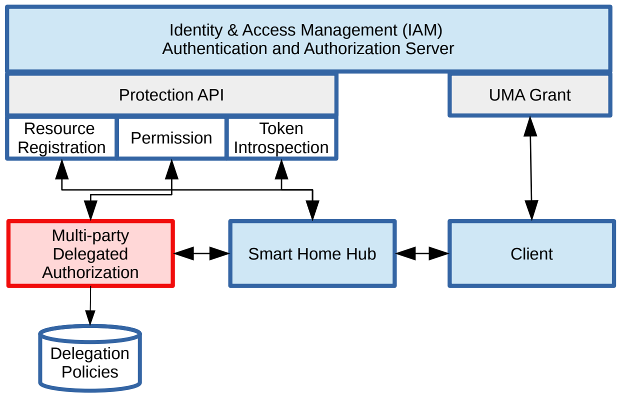

Fig. 4. Policy-based delegated resource ownership and access control

scopes with such fine-grained resource definitions. However, in other scenarios, it may be necessary that all, a majority or just one of the resource owners needs to grant authorization before access is finally permitted to the requesting party.

To handle multiple resource owners in a delegated authorization context, we add an additional building block to deal with delegation. As depicted in Fig. 4, a new component – implemented as a microservice – intervenes in step 3 and 4 of the UMA 2.0 protocol to identify the relevant permissions and scopes and return a permission ticket to the resource server and client. This way, the same resource can be protected with different UMA compliant authorization servers, and the *Multi-party Delegated Authorization* microservice is able to redirect to the proper authorization server based on the information provided by the client in the X-Resource-Owner custom HTTP header, as depicted below.

```
curl -X GET -H "X-Resource-Owner: bob" \
  https://example.com/smarthomehub/smartlock/unlock
```

```
GET /smarthomehub/smartlock/unlock HTTP/1.1
Host: example.com
X-Resource-Owner: bob
...
```

The above example is a client request to the resource server without the RPT, but with a reference to Bob as one of the resource owners that is in charge of authorizing the request. The resource server includes the custom HTTP header when interacting with the *Multi-party Delegated Authorization* microservice. The latter has delegation policies to manage the authorization server – and hence Protection API – per resource owner. Based on these policies, the permission ticket returned by the resource server will have instantiated the correct <host> of the authorization server of Bob:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: UMA realm="smarthomehub",
  as_uri="https://<host>/auth/realms/smarthomehub",
  ticket="01234567-89ab-cdef-0123-456789abcdef"
```

The client now has a permission ticket and the location of the authorization server of resource owner Bob. With this information, the client knows the token endpoint to send an authorization request.

The above example depicts delegation per resource owner – in this case user Bob – but the proposed solution is not limited to HTTP header-based delegation. The delegation policies in Fig. 4 may define a default authorization server when the X-Resource-Owner header is not available, or base its decision based on the information provided in any other header, bearer token or request parameter. In our proof-of-concept implementation, the *Smart Home Hub* microservice invokes the *Video Door Bell* microservice to capture the person in front of the door, and have that person identified by the *Face Recognition* microservice. Rather than delegating to an authorization server based on the resource owner, we now delegate based on the requesting party by having the policies in place to match resource owners and requesting parties.

The actual language of the delegation policy is independent of the protocol. Our proof-of-concept implementation uses the OPA policy language[6], as illustrated below for the smart lock use case. The data.json document below describes the resources, the resource owners and delegates per requesting party. For simplicity, we assume that every resource owner only uses a single authorization server.

```
{
  "resource_owners": [{
      "id": "bob",
      "name": "Bob",
      "authz_server": "iam1.com"
    },
    {
      "id": "alice",
      "name": "Alice",
      "authz_server": "iam1.com"
    },
    {
      "id": "eve",
      "name": "Eve",
      "authz_server": "iam2.com"
    }
  ],
  "requesting_party": [
    { "id": "bob", "delegate": ["bob","alice"] },
    { "id": "alice", "delegate": ["bob","alice"] },
    { "id": "eve", "delegate": ["eve"] },
    { "id": "charlie", "delegate": ["bob"] }
  ],
  "resources": [
    {
      "id": "smartlock",
      "uri": "**/smarthomehub/smartlock/",
      "owners": ["bob", "alice", "eve"]
    },
    {
      "id": "facerecognition",
      "uri": "**/smarthomehub/facerecognition/",
      "owners": ["admin"]
    }
  ]
}
```

The following example.rego declarative delegation policy will infer the proper authorization server endpoint for a given subject and resource.

[6]https://www.openpolicyagent.org

```
package opa.example

import data.resource_owners
import data.requesting_party
import data.resources

endpoint[[subject,resource,authz_server]] {
  subject = requesting_party[s].id
  resource = resources[r].id
  resources[r].owners[_] = owner
  delegate = requesting_party[s].delegate[_]
  owner = delegate
  resource_owners[o].id = owner
  authz_server = resource_owners[o].authz_server
}
```

The query below executed at the OPA command line will find the authorization server for the requesting party ”charlie” aiming to access the resource ”smartlock”. The actual proof-of-concept microservice implementation interacts with the REST interface of the OPA policy engine, rather than the command line.

```
# opa run data.json example.rego
OPA 0.10.7 (commit 0f39c27e, built at 2019-04-09T00:29:15Z)

Run 'help' to see a list of commands.

> import data.opa.example.endpoint
>
> endpoint[x][0]="charlie"; endpoint[x][1]="smartlock"
+-----------------------------------+
|                 x                 |
+-----------------------------------+
| ["charlie","smartlock","iam1.com"] |
+-----------------------------------+
>
```

The delegated authorization server is ”iam1.com”, because Bob is one of the resource owners and also the delegate for the requesting party Charlie, and iam1.com is the declared authorization server for delegate Bob.

E. Cross-domain multi-party access control with delegated evaluation of authorization policies

The fourth security requirement of section III.B states that the delivery guy can only enter the building when authorized by both the delivery company and either Bob or Alice. As the delivery guy can only be identified by his employer, the *Smart Home Hub* microservice should invoke the *Face Recognition* microservice of the package delivery company, and must be authorized to do so. Depending on whether the delivery guy can be identified based on the picture created by the video door bell, the smart lock may grant access.

In this cross-domain access control scenario, multiple authorization decisions are taken but with a subtle difference regarding the identities of the requesting parties involved:

- **Requesting party is known:** The authorization server of the package delivery company implements a policy to control which third party can access its face recognition service. Due to Know Your Customer (KYC) compliance, the company can encode claims about the requesting party Bob in a JWT token. Using the OAuth 2.0 protocol, the Smart Home Hub can access the company's face recognition service on behalf of Bob.
- **Requesting party is not known:** The authorization server of the smart home hub does not know the delivery

guy. It must rely on identity brokering or claims gathering protocols to evaluate its authorization policies (see previous examples) and grant the delivery guy (i.e. the requesting party) access to the smart lock microservice. Our proof-of-concept implementation relies on certified claims for privacy reasons. The claim only states whether the requesting party is an employee of the package delivery company, but not the actual identity of the requesting party which is only known to the company. The latter can provide a proof-of-authentication, but collaboration between the domains is required if the identity should also be revealed to the resource owner.

For performance and security reasons, the package delivery schedule – the day and times in between the package is expected to be delivered – is encoded as an authorization policy as metadata in the JWT token.

```
{
  "alg": "HS256",
  "typ" : "JWT",
  "iss": "https://packagedelivery.com",
  "name": "bob",
  "exp": 1551567600,
  "iat": 1551308400,
  "app_metadata": {
    "policy": {
      "resource": "/facerecognition/identify",
      "actions": [ "post" ],
      "effect": "permit",
      "conditions": [
        ">": { "time": 1551531600000 },
        "<": { "time": 1551549600000 }
      ]
    }
  },
  "user_metadata": {
    "role": "delivery_guy",
    "package": "7ab7f454-e082-493b-a4f7-1d622b84b02f"
  }
  ...
}
```

The above JSON Web Token (JWT) includes an authorization policy - encoded as metadata – granting access to the face recognition service on March 2, 2019 between 2PM and 7PM (encoded in milliseconds since epoch).

The advantage of this delegated authorization policy evaluation is two-fold:

1) **Performance:** The *Smart Home Hub* microservice will not invoke the company's *Face Recognition* microservice beyond these time slots, as authorization will not be granted anyhow.
2) **Security:** Beyond the time slots, access to the unlock interface of the *Smart Lock* microservice will be immediately denied without even attempting to remotely identify the person at the door.

Note that this simple time-based authorization policy could also be implemented using the standard nbf and exp JWT fields to indicate the time on which the JWT will start to be accepted for processing and the expiration time. However, the JSON structure of the authorization policy support more sophisticated access rules and conditions.

Access to the current distance and fine-grained location of the delivery guy is implemented in a delegated manner as well. It is time-boxed on both sides of the microservice federation for performance reasons, and with an additional authorization policy at the company's side for security/privacy reasons.

```
var usage_per_min_stats = 3;
var destination = [51.5074, -0.1277];
var current_location = [51.5184, -0.18677];

function distance(from, to) { ... }

function authz_filter() {
    if (usage_per_min_stats > 5) {
        return null;
    }
    var d = distance(current_location, destination);
    if (d > 10) {
        return d;
    }
    return current_location;
}
```

This JavaScript-based authorization policy is a simple example to illustrate (a) granting *selective consent* to the current fine-grained location of the delivery guy when within a certain distance to the destination, and (b) enforcing *usage control* on the requesting party to invoke the service at most 5 times per minute.

F. Securing the multi-party delegated authorization service

The advantage of the proposed multi-party delegated authorization service is that it is completely transparent towards (a) the UMA 2.0 protocol and (b) the policy language used for multi-party delegation to redirect to different UMA 2.0 compliant authorization servers per resource owner. In that sense, the *Multi-party Delegated Authorization* microservice acts as a policy-driven authorization gateway.

However, access to this service must be secured as well, as not every individual should be authorized to change the resource owner specific data files and delegated authorization policies. As these delegated authorization resources are exposed through OPA's RESTful interfaces, we leverage the same UMA protocol to authorize access to the policy administration interfaces by means of a UMA compliant IAM instance (in our proof-of-concept: Keycloak). Our *Multi-party Delegated Authorization* microservice is implemented as a Spring Boot companion application for the OPA policy engine. The RESTful interfaces of the OPA policy engine are not directly accessible by other clients, but only indirectly through our microservice. To realize this, the latter follows the Sidecar design pattern for microservices and its RESTful interfaces are resources that are protected by the Keycloak IAM.

IV. Evaluation

This section reports on various qualitative and quantitative evaluation metrics of our solution, identifying performance and impact trade-offs.

A. Experimental setup

The scenario in section III has been implemented as a collection of Java-based Spring Boot 2.1 microservices using the Spring Cloud Greenwich.SR1 framework[7].

[7]https://cloud.spring.io

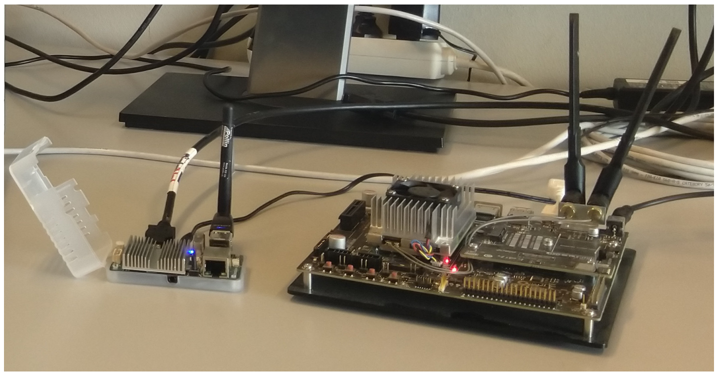

Fig. 5. The ODROID-U3 mini-board and NVIDIA Jetson TX2 development board as edge devices.

- **Cloud Server:** The microservices of the package delivery domain were deployed in a Docker 18.06 container, running on top of Ubuntu 18.04 on a Dell PowerEdge R620 server with an Intel Xeon CPU E5-2650 16-core processor running at 2.00GHz (with hyper-threading enabled resulting in 32 virtual CPUs) and 64GB of memory.
- **Edge Device 1:** The smart home microservices were deployed on a fairly old ODROID-U3 miniboard from 2014 with 2GB of memory and a Cortex-A9 quad-core 1.7GHz CPU (see left device in Fig. 5) and running a 32-bit ARM version of Ubuntu 18.04. They communicate with the remote microservices in the Docker container over a WiFi connection.
- **Edge Device 2:** A similar edge setup was created with a NVIDIA Jetson TX2 development board having a dual-core Denver 2 64-bit CPU and quad-core ARM A57 complex 8 GB of memory, and a Pascal-family GPU with 256 cores (see right device in Fig. 5) and running a 64-bit ARM version of Ubuntu 18.04. The GPU is more efficient to handle the deep learning-based face recognition at the edge of the network. Network connectivity is the same as for the ODROID-U3 edge device.

The core UMA 2.0 implementation is offered out-of-the-box by RedHat's Keycloak 5.0[8] open source IAM solution and its Spring Boot authorization service adapters. In order to evaluate Rego policies on edge devices, the OPA policy engine (version 0.10.7) was cross-compiled for both 32-bit and 64-bit ARM-based edge devices.

When requesting access to a protected API offered by one of the Spring microservices, the Spring Boot authorization service adapter embedded in the microservice acts as the *Policy Enforcement Point* by intercepting incoming requests and enforcing the authorization policies. The decision to grant or deny access is returned by Keycloak which acts as the *Policy Decision Point*. As Keycloak 5.0 by default only supports authorization policies defined in JavaScript, we implemented a Javascript proxy to delegate the policy evaluation and decision to the external OPA policy engine. The latter becomes the *Policy Administration Point* of Rego policies and runs as a dedicated microservice.

[8]https://www.keycloak.org

B. Edge performance results

In Fig. 6, we measure the latency of token and policy evaluation when invoking protected APIs on a Spring Boot microservice. The microservice as well Keycloak and the OPA policy engine are all deployed on both edge devices and on a resource rich cloud environment for comparing the response times. The OAuth token response time is the end-to-end latency of a Google Chrome browser running on a client device and invoking an HTTP POST request on the OAuth REST endpoint of Keycloak. For the sake of simplicity, we use the OAuth 2.0 Password grant type to exchange a user's credentials for an access token. The RPT Token response time measures the end-to-end latency to exchange the OAuth access token with an RPT Token. During this step the relevant Rega policies are also evaluated by the OPA policy engine. The response time of the OPA policy engine is depicted in the third bar and is at least an order of magnitude smaller than the RPT Token response time. It is clear that the most heavyweight component is Keycloak.

Nonetheless, even with 2GB of memory, the ODROID-U3 miniboard was able to run Keycloak 5.0 and the OPA 0.10.7 policy engine, next to several other Java-based Spring Boot microservices. This allows for the authentication and authorization services to be shifted from the cloud towards the edge. Although one may expect a lower overall latency by running in the edge and avoiding round trip times to the cloud, the limited computational capabilities of the ODROID-U3 miniboard meant that any gains in terms of reduced networking latency succumb to the longer request processing times for Keycloak when executed on the miniboard in the edge. Nonetheless, with identity brokering and replication in place, the edge device can serve as a fallback whenever connectivity to the cloud is temporarily not available. However, the FaceNet convolutional neural network for face recognition [18] caused significant peak loads on the ODROID-U3 miniboard during which the other microservices deployed on the same device were slowed down too. These high CPU peak loads are caused

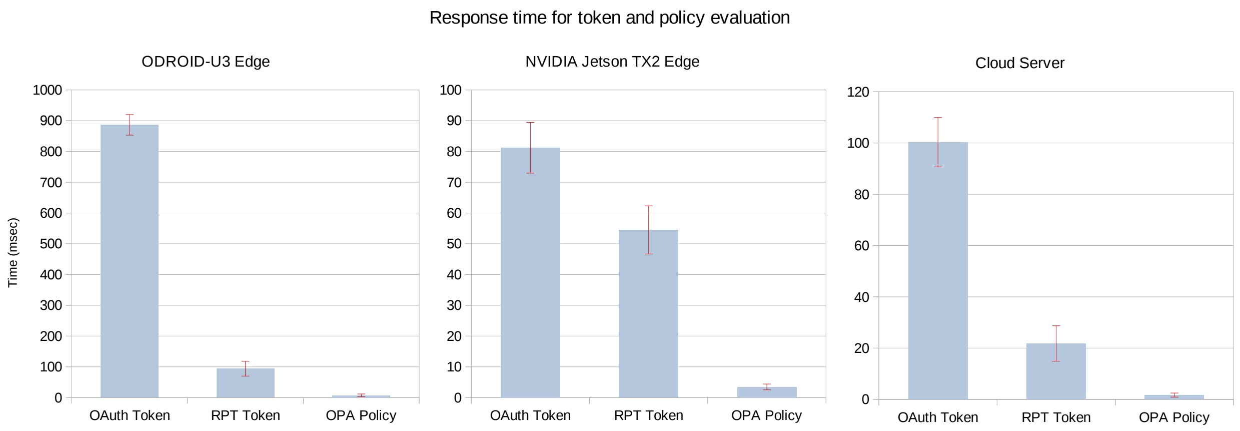

Fig. 6. Average response time and standard deviation for token and policy evaluation on edge devices and cloud server

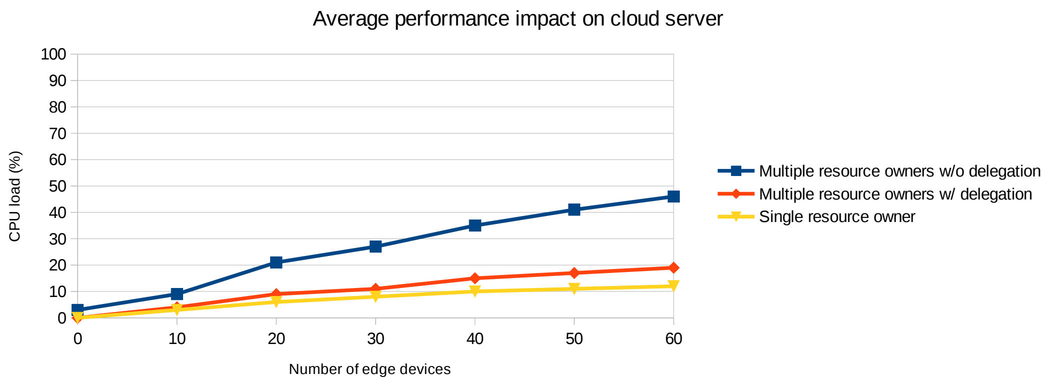

Fig. 7. Impact of delegated multi-party access control policy evaluation under a growing number of edge devices with microservices having protected APIs

by the fact that the neural network behind the face recognition algorithm is evaluated by the CPU rather than a hardware accelerator (e.g. a CUDA enabled GPU).

As the NVIDIA Jetson TX2 development board has more computational resources, and a Pascal-family GPU to evaluate the FaceNet neural network, the performance impact of the face recognition component on the other microservices was less significant. This shows that it is feasible to (1) push face recognition completely towards the edge, or (2) let the edge device compute the facial embeddings and only submit an 128-dimensional vector – rather than the full camera picture – to a remote cloud service for recognition by another party in the federation. Such an approach is only feasible of both parties rely on the same pre-trained FaceNet neural network.

To ensure a fair comparison for both edge devices, the face recognition microservice was hosted on a separate node to not skew the performance comparison in Fig. 6.

C. *Cloud performance results*

As part of our cloud-oriented evaluation, we will assess the performance overhead of our multi-party delegated authorization against a single-owner scenario (i.e. baseline UMA 2.0), and this for a growing number of edge devices (i.e. one is a physical device and the other devices are simulated on a desktop machine). Fig. 7 depicts the performance evaluation results of 3 different configurations:

- **Single resource owner**: This scenario assumes there is only one owner per resource for which the authorization server is known. The configuration resembles the default UMA 2.0 protocol.
- **Multiple resource owners with delegated policy evaluation**: This scenario assumes multiple resources owners, and requires redirection towards our *Multi-Party Delegated Authorization* microservice to select the correct authorization server.
- **Multiple resource owners without delegated policy evaluation**: Contrary to the previous configuration, the edge devices do not evaluate the authorization policy embedded in the JSON Web Token to minimize the impact on the remote service.

The performance results show that there is a small but acceptable relative performance overhead – varying between 10 and 25% – for the scenario with multiple resource owners. In practice, we expected an even lower overhead as the redirection by the *Multi-Party Delegated Authorization* microservice is executed on the edge device, and not in the cloud. However, this configuration creates more concurrent network connections from different clients to the cloud, with a connection pooling that is most likely less effective. The round trip times of the authorization protocol (as depicted in Fig. 3) are not significantly different when delegation is used due to the latency already imposed by the wireless network.

When partial policy evaluation on the edge device is disabled, the performance impact is much more significant due to authorization requests for the delivery guy's location being denied (e.g. requests submitted too frequently or beyond the expected delivery times). Note that the absolute values are specific to this particular use case and the complexity of the different authorization policies and policy engines.

V. Conclusion and future work

Fog and edge computing are two paradigm shifts that give rise to the decentralization of the deployment of cloud services. As a consequence, new security and privacy challenges emerge, one of which is the administration, evaluation and enforcement of access control policies for cloud-enabled edge deployments where (micro)services must cooperate at the edge and with services in the cloud. As the number of stakeholders in such collaborative and federated deployment environments increases, delegating the management of access control decisions – also to non-security experts – is not straightforward, especially when multiple parties are involved in access control decisions.

To ensure that user-centric access control remains sustainable in such complex service delivery models, we presented and evaluated a dynamic granular access control solution on top of RedHat's Keycloak 5.0 Identity and Access Management (IAM) solution and the User-Managed Access (UMA) 2.0 specification, as well as other policy-based authorization frameworks. In particular, our multi-party policy-based access control solution can handle multiple owners per resource and delegate the policy authorization, and this with a limited relative performance overhead.

As part of future work, we will research to what extent our multi-party policy-based access control system can be enhanced to also coordinate the secure deployment and orchestration of microservices in federated cloud-edge-fog environments.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven, and by imec through ICON FUSE. FUSE is a project realized in collaboration with imec, with Barco, e-BO Enterprises and Axians as project partners and with project support from VLAIO.

References

[1] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media, Inc., 2015.

[2] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[3] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/3147213.3147229

[4] A. C. Squicciarini, S. M. Rajtmajer, and N. Zannone, "Multi-party access control: Requirements, state of the art and open challenges," in *Proceedings of the 23Nd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '18. New York, NY, USA: ACM, 2018, pp. 49–49. [Online]. Available: http://doi.acm.org/10.1145/3205977.3205999

[5] L. González-Manzano, A. I. González-Tablas, J. M. de Fuentes, and A. Ribagorda, "Cooped: Co-owned personal data management," *Computers & Security*, vol. 47, pp. 41 – 65, 2014, trust in Cyber, Physical and Social Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404814000960

[6] D. Fett, R. Küsters, and G. Schmitz, "A comprehensive formal security analysis of oauth 2.0," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1204–1215. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978385

[7] E. Maler, "Extending the power of consent with user-managed access: A standard architecture for asynchronous, centralizable, internet-scalable consent," in *2015 IEEE Security and Privacy Workshops*, May 2015, pp. 175–179.

[8] E. Rissanen, "eXtensible Access Control Markup Language (XACML) Version 3.0," OASIS Standard, January 2013. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[9] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC)," in *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, ser. ABAC '16. New York, NY, USA: ACM, 2016, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2875491.2875496

[10] T. Sandall, "Open policy agent," 2019. [Online]. Available: https://www.openpolicyagent.org/

[11] D. Hardt, "The OAuth 2.0 authorization framework," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, RFC 6749, Oct. 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt

[12] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," Tech. Rep., 2015.

[13] M. Schwartz and M. Machulak, *User-Managed Access*. Berkeley, CA: Apress, 2018, pp. 267–299. [Online]. Available: https://doi.org/10.1007/978-1-4842-2601-8_8

[14] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 65:1–65:45, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3007204

[15] M. Uriarte, J. Astorga, E. Jacob, M. Huarte, and M. Carnerero, "Expressive policy-based access control for resource-constrained devices," *IEEE Access*, vol. 6, pp. 15–46, 2018.

[16] H. Jiang and A. Bouabdallah, "Jacpol: A simple but expressive json-based access control policy language," in *Information Security Theory and Practice*, G. P. Hancke and E. Damiani, Eds. Cham: Springer International Publishing, 2018, pp. 56–72.

[17] A. D. Brucker and H. Petritsch, "Extending access control models with break-glass," in *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '09. New York, NY, USA: ACM, 2009, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1542207.1542239

[18] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering." *CoRR*, vol. abs/1503.03832, 2015. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1503.html#SchroffKP15

[19] W. Li and C. J. Mitchell, "Analysing the security of google's implementation of openid connect," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 357–376. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_18