Rethinking Authorization Management of Web-APIs

Bojan Suzic
IEEE Member
Den Haag, Netherlands
bsuzic@ieee.org

Milan Latinovic
TimeTac GmbH
Graz, Austria
milan.latinovic@timetac.com

***Abstract*—Service providers typically utilize Web APIs to enable the sharing of tenant data and resources with numerous third party web, cloud, and mobile applications. Security mechanisms such as OAuth 2.0 and API keys are commonly applied to manage authorization aspects of such integrations. However, these mechanisms impose functional and security drawbacks both for service providers and their users due to their static design, coarse and context insensitive capabilities, and weak interoperability. Implementing secure, feature-rich, and flexible data sharing services still poses a challenge that many providers face in the process of opening their interfaces to the public.**

To address these issues, we design the framework that allows pluggable and transparent externalization of authorization functionality for service providers and flexibility in defining and managing security aspects of resource sharing with third parties for their users. Our solution applies a holistic perspective that considers service descriptions, data fragments, security policies, as well as system interactions and states as an integrated space dynamically exposed and collaboratively accessed by agents residing across organizational boundaries.

In this work we present design aspects of our contribution and illustrate its practical implementation by analyzing case scenario involving resource sharing of a popular service.

I. Introduction

The value of data produced and stored at a multitude of interconnected devices strongly depends on the capacity to manage its sharing in a flexible and overseeable way. Web API represents one of the ordinarily applied mechanisms to expose data and services [1]. Being initially related to the realm of cloud and web services, Web APIs now exhibit a more pervasive context with IoT [2] and mobile devices [3].

Traditional solutions for access control of Web APIs tend to deal with a restricted subset of *authorization* related aspects [4]. As one of the examples, OAuth 2.0 [5] establishes the interactions for communicating resource requirements and credentials among different systems. *Access scopes*, as its means for conveying permissions, are built as opaque data references and specified in natural language. It is thus not possible for a machine to understand and autonomously derive or adjust the meaning of a provided scope. OAuth furthermore does not deal with a model to establish or manage security policies, nor it knows anything particularly about the structure of APIs and exposed data it is intended to protect.

XACML [6], on the other hand, focuses on policy-related aspects of authorization control, providing a comprehensive and extensive framework for definition and evaluation of security policies. In the context of Web APIs, the primary link between API or data structure and XACML policies is an *attribute*, whose meaning is arbitrary determined by developers and interpreted strictly in the domain of a particular environment. Driven by *intra-enterprise* use scenario, XACML does not support clients in defining their security requirements, nor provides a framework on how to specify and apply security policies beyond a single system and domain.

UMA [7] adopts and extends cross-system interactions from OAuth 2.0 and further introduces an entity functionally similar to XACML's *policy decision point* to allow externalizing access decisions to user's domain [8]. UMA is, however, neutral with regards to the structure of APIs, protected data, or policies. Their representation and use beyond a single domain are thus left to implementing entities to decide.

The consequences of such design choices are manifold. Incomplete aspects of existing frameworks leave implementing organizations with too many gaps in the process of exposing their services and tenant resources to third parties. To support their customers in reaping the benefits of a data-driven economy [9], organizations typically need to evaluate various possibilities and decisions related to strategical, architectural, and security aspects of data sharing. While many services rely on similarly conceptualized Web APIs [1], [10], many of these entities have to independently and repetitively face similar sets of questions and decisions in practically (re-)implementing data sharing and authorization capabilities.

One of the consequences of incomplete building blocks is a widespread prevalence of APIs with similar rudimentary, shallow and inflexible security capabilities [11].

For consumers, suboptimal solutions imply less control, basic security, and minimal flexibility in managing authorizations. For providers, this also means increased implementation costs and reduced range of provided functionalities. For both of them, reduced security features may negatively impact the perceived value of distributed data, resources and services.

II. Problem analysis

Access scopes in OAuth 2.0 allow service providers to define abstract ranges of coarse-grained permissions as concepts that can be referenced across interacting entities to request and provide authorization consents. OAuth 2.0 specification introduces scopes as parameters with the purpose to (1) allow clients to specify the degree of their access requests, and to (2) enable providers to inform the clients about the range of accepted or provided permissions [5]. The permissions implicitly communicated within the OAuth 2.0 scope are

978-1-7281-4657-7/20/$31.00 ©2020 IEEE

associated with each access token provided to the clients for single or repetitive accesses [12], [13].

OAuth 2.0 defines the structure of the scope parameter as a list of space-delimited, case-sensitive strings, whose inherent permission extent is predefined by the provider and typically left unchanged during the lifecycle of an API version. Multiple of these strings (scopes) can be combined in scope parameter to express a combined range of access permissions.

Besides recommending that service providers should document their scopes, OAuth 2.0 specification does not provide any additional details that would allow dereferencing of scopes or establishing of cross-system interoperability at a higher level, beyond the *opaque* strings and *hard-wired* logic [14].

The mechanism involving access scopes allows authorized clients to perform unlimited accesses, restricted only by abstract, unilaterally defined and non-inferrable scope coverage. As scopes relate to concepts, rather than to data or object instances, their use in controlling access based on dynamic properties or structure of protected resources is practically not feasible. We can observe this property from the example presented in Section V-B, where the scopes allow restricting accesses using only a range of activities predefined by the service provider, which stay static during the lifecycle of related API version.

A similar limitation applies to the specification of dynamic data transformations, which are necessary to implement advanced privacy and legal requirements. Notably, the range of exposed data rarely equals to exact requirements of data sharing use case. Rather, clients usually get a broader degree of permissions than needed for the successful execution of their use case. Moreover, the overall mechanism behind access scopes does not accommodate the derivation of contextual confinements, which allow access control based on a range of contextual parameters relating to the resource or environmental conditions. These limitations stem from the tight expressive capability, lack of scope structure and its relational detachment from systems and the environment.

In our previous contribution, we have analyzed the application of authorization mechanisms on RESTful APIs on a large scale. For further details and identified issues, we refer the reader to that work [11].

In the rest of this work, we first introduce the high-level design and architectural aspects of our solution. We then provide additional details on system interactions, underlying capabilities, and internals. In the subsequent chapter, we demonstrate the application of our framework based on the motivational case scenario using a popular web service, which is followed by the discussion of results and related work.

III. Design and Operation

To address the deficiencies of currently applied solutions we propose conceptual framework and implement demonstration prototype to demonstrate and examine its applicability.

A. Design goals

Our goal is twofold. First, we aim to support companies in integrating multilateral data sharing within their services by providing a fully *externalized*, *pluggable*, *flexible* and *context-sensitive* authorization management solution that integrates well with existing protocols and technologies.

Following the drawbacks of existing approaches, we intend to establish API-aware security policies that allow dynamic evaluation of structural parts of protected resources, interaction states, and environmental properties. Moreover, the policies should allow the client or state-driven online adjustment of information footprint of shared resources.

As second, we want to enable users to independently manage access to their resources across different service providers using a single, integral framework that allows rich and granular expressivity of security goals and their enforcement at *user* or *provider* premises. It should improve the balance of power in the current environment, where service providers unilaterally decide on degree of supported security capabilities in sharing of their services, and typically on the lower level [11].

B. Unified situational model

For this purpose, we establish the framework that applies a holistic approach by integrally considering and interrelating different building blocks relevant for authorization management of Web APIs. In vertical integration plane, our solution couples API structure, representations of data flow interactions, and authorization models. In the horizontal integration plane, we facilitate interoperability and reuse of security controls among diverse subjects, which reduces implementation friction and allows a higher degree of flexibility and capabilities in security management.

To enable integral authorization management we leverage the concept of *knowledge graph* to interconnect representations of web service resources, their organizational structure, requests, and interactions occurring in the system with entities describing security functionalities.

This graph uses the model of a web service as a starting point to define *unified situational model*. Our conceptual basis for the structural description of a web service relies on broadly applied concepts across RESTful deployments in the practice [15]. We hence adopt the view of a RESTful service based on *resource* and *operation* and reuse these concepts as a basis for Web API descriptive building block in our model.

Formally, the definition of the underlying structure of the knowledge graph relies on three sets. The first set T defines a range of usable *types*, whose *instances* are represented as the members of the second set E. T thus establishes a selection of entities that form a *conceptual basis* to build the knowledge graph. Instantiations of entities from this set are used to describe Web APIs, interactions between systems, and applicable security controls. Sets T and E can be characterized as *T-Box* and *A-Box* knowledge, respectively [16].

The third set R consists of binary, asymmetric *relationship labels*, which are applied to instantiated entities in the API model and express their behavioral or structural interdependencies.

The instance of an *integrated model* can be represented as directed, multi-labeled graph $G(V, A)$, with vertices consisting

of elements from E and edges connecting these elements with relationships defined in R.

Types and relationships defined in sets T and R can be further refined and separated in subsets T_n and R_n, where n represents a particular aspect or subdomain of the integrated model. In our framework, we group related types and relationships in different subdomains to allow flexible and pluggable representations of the knowledge graph. This is beneficial, for instance, in providing support for different security mechanisms applicable in a particular case scenario, such as OAuth 2 and API keys. It also allows using different representations of web service structure, allowing for greater flexibility and customization of the framework. The instance of an *integrated model* can be represented as directed, multi-labeled graph $G(V, A)$, with vertices consisting of elements from E and edges connecting these elements with relationships defined in R.

C. Core vocabularies

To facilitate the definition and the practical application of our integrated model-based approach, we establish DASP (DAta Sharing and Processing) framework consisting of a set of vocabularies separated into functional layers. The primary purpose of this framework is to provide a structured and modular conceptual basis for descriptions of entities and processes that take part in service-based interactions. This basis enables the instantiation of a view that unifies both service and data descriptions, security policies, and interaction elements to allow expressive specification and enforcement of security policies beyond a single service.

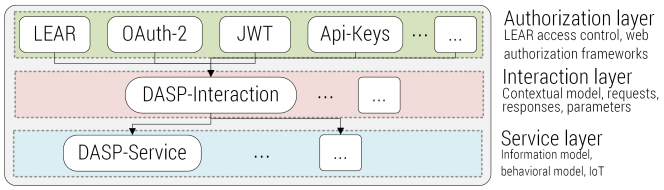

Fig. 1. Layers and vocabularies in DASP framework

DASP framework considers the existence of three primary actors: (1) service provider (SP) hosts resources, provides services or performs data processing, (2) resource owner (RO) or user (tenant) owns resources hosted at SP or subscribes for its services, and (3) client (C) accesses resources or consumes services at SP on behalf of user or according to its policies.

We model the domain using three layers. *Service layer* provides the view of information and behavioral model of a web service. *Interaction layer* provides vocabularies that support describing the interaction that occurs among the actors in the system, including requests, responses, contextual properties, and resource (transformational) restrictions. The third layer allows the definition and integration of different access control models or web authorization frameworks. While *DASP-Service* and *LEAR* are primary vocabularies used in this work, the modular nature of the framework allows definition and use of different vocabularies that may better suit particular use case.

DASP vocabularies are specified in a compact, lightweight manner, following *self-descriptive* and *bottom-up approach* [17]. Due to the size of vocabularies and space restrictions, we refer to a web location that provides vocabularies in human and machine-readable forms [18].

D. Externalized authorization

The core functional entity in DASP framework is Δ_{gw}, a modular middleware component deployed in front of service provider API that jointly integrates both functionalities of management and enforcement of authorizations.

Δ_{gw} name relates to two characteristics of this component. *Gateway* represents its functional role in the network to accept, examine, and translate client requests. *Delta* as the second aspect is related to the active role in dynamically adjusting the information footprint of client requests and service responses. Namely, Δ_{gw} implements a range of functionalities that allow enforcement of *the principle of least privilege* [19] by dynamically transforming the content of incoming or outgoing messages in a context-sensitive manner.

In terms of enforcement, Δ_{gw} supports *user-centric* and *provider-centric* mode. In provider-centric deployment, Δ_{gw} executes as a part of SP's infrastructure, controlled and administered by its authoritative entity. This typically implies a coupling with (single) SP service. In user-centric deployment, one Δ_{gw} can intercept and translate requests for resources associated with a tenant domain of a single user, the resources being hosted at several different backend SPs.

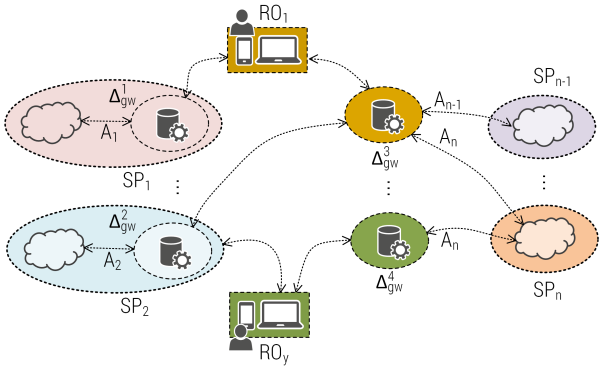

Fig. 2. Management and deployment of Δ_{gw}

Figure 2 depicts two deployment modes from the perspective of the administration of security goals. The left part of the figure shows provider-centric deployments that enforce security policies over the tenant resources hosted at a service provider. The right part of the figure shows the user-centric deployment, where one middleware instance can manage resources at several SPs for a single user. In this case, Δ_{gw} instances can be deployed at user premises or third party providers, where the latter may be delivered under *Security as a Service* model.

From the perspective od clients accessing user resources at a service provider, the interfaces are transparent and conform to the API contract of a target service provider. From the

user (tenant) perspective, the presented setup allows for greater flexibility in managing the security of resource sharing as the user is able to use a single, unified, cross-provider framework to control its resources scattered at different clouds.

IV. Authorization management workflow

In this chapter, we describe the phases in the overall process of managing access authorizations over user resources at a web service. We leverage the model-based approach that relies on lightweight, modular, and self-contained semantic descriptions of services, processes, and policies. We integrally employ and use these descriptions across different subjects involved in the authorization workflow in the end-to-end manner. Considering its context and domain, each subject and phase of the workflow reuse and augment a model originating from the initial service description. Figure 3 gives an overview of this process and its key aspects. In the following sections, we leverage examples from running case scenario to describe each of these key aspects and present the phases of our model-based approach.

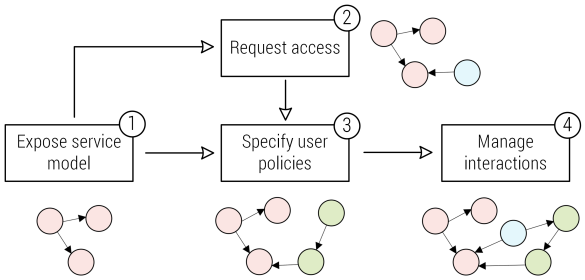

Fig. 3. Phases of authorization management process

A. Defining service model.

To enable the specification and enforcement of user security goals, it is first necessary to define a model that represents the service and its exposed resources which we need to protect.

A service model is described by expressing its structure as a directed graph using instantiations of concepts established in *DASP-Service* vocabulary as nodes, and interrelating them using supported properties as links (see Figure 1).
This model is then retrieved and its entities reused by parties involved in the management workflow for the purposes including: 1) expressing permissions and service resources requested by clients, 2) specifying security goals of resource owners, and 3) enforcing security goals in access infrastructure.

An example excerpt from the specification of a service model is depicted in Figure 4. This fragment reflects the second API endpoint introduced in Section V-B, which corresponds to the operation of email retrieval exposed by Gmail RESTful API.

In this specification, we observe node labeled as *Gmail service*, which is an instance of *Service* class (S) from *DASP-Service* vocabulary and represents a root concept in describing services. The property labeled with *hasAction* defined in the same vocabulary connects *Gmail service* node with the node labeled as *Retrieve email* (A). This relationship is used to express that *Gmail service* exposes an action to *Retrieve email*.

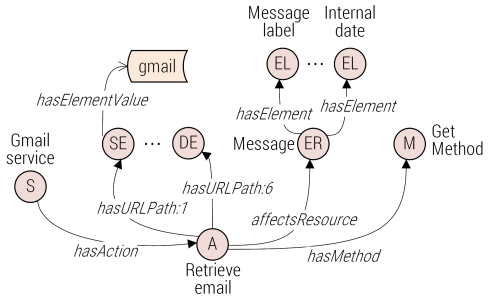

Fig. 4. Service model fragment - action and resource structure

The links to other nodes further specify the action's properties. This way, the instance of *HTTP GET method* (M) labeled as *Get method* indicates that *Retrieve email* action receives API calls using `HTTP GET` method.

Further entities on the figure, referenced under *hasURL-Path:n* properties, indicate the elements of the API endpoint path of the action. We can observe the use of classes of *Static path element* (SE) and *Dynamic path element* (DE). The former relies on a string value to represent a relative fragment of the URL endpoint. The later serves to indicate API endpoint fragments whose values may change depending on the context. This case is depicted on the figure as a dynamic element without predefined value, corresponding to the sixth element of the second API endpoint shown in Section V-B.

The excerpt in Fig. 4 further shows two instances of *Element* classes (EL) connected with a node using *hasElement* property. These instances refine the description of the *Message* resource (ER) exposed (or handled) by the corresponding action *Retrieve Email*. Each of these elements represents an entity in a hierarchical structure of a complex data object. This enables dealing with representations common in RESTful services, which typically rely on JSON or XML data formats.

The capability to granularly represent data structures allows fine-grained specification of security goals, as each of these elements can be directly referenced in security policy. The enforcement of such goals, however, requires mechanisms to enable automated retrieval and evaluation of nested elements from API resources. This is enabled by using instances of *Element extractor* class (EE) to define extraction rules for resource entities in multiple data formats. An instance of such a rule is illustrated in Fig. 5 with *MsgLabelExtractor* node, which extends the initial service model with the instruction on element retrieval using *JSON Path* expressions [20]. The rule for other data formats is analogously defined by relying on properties corresponding to other languages.

B. Requesting client access

The definition of user security goals can generally take place in two ways. In a first instance, users may establish security goals in a *proactive* manner, by *a priori* specifying security rules for still unknown access requests that potentially may occur. XACML [6] follows this approach by separating policy definition from client requests. If there is a basis that involves a client in an *a priori* definition of rules, this

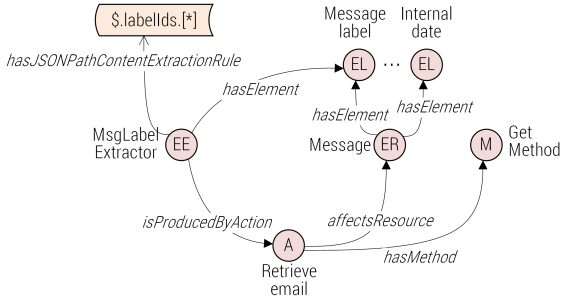

Fig. 5. Service model fragment - retrieval instructions

basis is communicated in an *out-of-the-band* process. Alternative possibility follows a *reactive* approach, where clients, prior to performing access to a resource, request necessary permissions. This is usually performed using *interactive* or *backend* channels, allowing a resource owner to approve or revise requested permissions and integrate them as a part of a corresponding security goal. OAuth [5] web authorization framework applies this approach.

Our framework considers a hybrid definition of security goals that combines both of these approaches. While explicitly requesting access is seen as an optional step, it allows context-driven fine-tuning of the degree of API access capabilities in a *cooperative* manner that promotes adaptive balancing between utility for clients and security for resource owners. The cooperative aspect is further supported by a mechanism that allows clients to express not only their resource requirements but also the acceptable restrictions applicable over requests or resources.

To express its request, a client has first to retrieve the model of service and, based on entities and relations given in model descriptions, structure its request. The access request typically includes required *actions* or *resources*, and optionally may specify an acceptable degree of restrictions that are deemed as non-critical for the client's use case.

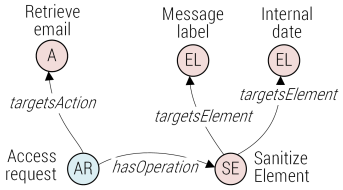

Fig. 6. Access request model fragment

Figure 6 provides an example of this specification considering the scenario from Section V-D. In this instance, the client specifies the request to access the action corresponding to email retrieval and declares that the sanitization of elements corresponding to the message label or internal date of emails is acceptable for its use case.

The specification from Figure 6 is dealt with three steps. In the first step, the client retrieves the service model, whose fragment is provided in Figure 4. Based on this model, the client forms a request by instantiating a node corresponding to *access request* (AR) class of *DASP-Interaction* and connects it with the required action from the service model. The access request illustrated in Figure 6 is additionally connected with the node of class *Sanitize Element* (SE). This instance is created by the client according to exposed capabilities in the service model and connected using properties with two elements contained in the structure of email resource, as shown in Figure 4. In such an arrangement, the client declares non-relevance of the content of referenced entities for its data processing.

C. Specifying and enforcing user policies.

Once the service model is defined and available, the resource owner can start formulating security policies that govern access to exposed resources. This definition is performed similarly as the specification of a service model - by instantiating the concepts established in the vocabularies of the framework. While our framework envisages support for several access control mechanisms, in this work, we rely on *LEAR* and *DASP-Interaction* vocabularies.

As specifying security policies is performed analogously as the description of a service model or an access request, we elaborate technical details of this aspect in the following chapter as part of a detailed use case study. Prior to transitioning to the analysis of use cases, however, we introduce the concept of a *unified model*, which is essential for the *enforcement* of user policies.

As depicted in Figure 3, each phase in the management workflow builds upon previous, phase-specific iterations of existing models. So the policies established in the third phase rely on service description defined in the first phase by referencing its entities. Specification of policies may optionally integrate client access requests as well. During the online client accesses, the model resulting from the third phase is further extended with the entities that describe intercepted and processed interactions in a system. Two main activities characterize the management of these interactions.

The first activity, the extension of the model, is performed during the monitoring and interception of events related to the protected service. At this point, the enforcement component instantiates descriptions of identified interactions and injects them in the existing graph. This step allows the enrichment of the initial model with entities that describe requests associated with service and policy resources. In this regard, the enforcement gateway builds and maintains a *unified model*, which contains the necessary information to continue with the subsequent process steps.

The second activity in the third phase relates to the policy decision process. This activity starts from the intercepted and modeled interactions by performing the exploration of the unified model to retrieve relevant data and relationships. The policy decision is derived based on identified security policies, relevant contexts, and then implemented at the enforcement gateway.

The following chapter illustrates the representative parts of this process through real-world use cases.

V. Case Scenario

In this chapter, we illustrate the application of our framework on a real-world use case scenario represented by the popular and broadly used Gmail service. We particularly focus on practical aspects of specifying and enforcing user security policies based on DASP framework and its components.

A. Personal data sharing using GMail

In recent years various applications emerged that extract information from unstructured email messages and integrate it into further commercial workflows and process automation. One of the prerequisites for this activity is the sharing of email messages with third-party services that periodically extract and process client data for further business workflow integration.

Our scenario considers the use of the freely available Google Mail (GMail) service for email hosting. Google exposes RESTful API [21] for email account management and relies on OAuth 2.0 [5] to authorize access for third parties. The sharing of email with external subjects typically relies on owner *consent* provided in one of the supported OAuth *flows*. This consent associates an accessing client, user's (tenant's) resources at GMail, and the *scope*, which describes the authorization extent, and implicitly defines the degree of client access permissions.

Resource sharing and data integrations with third parties generally rely on Web APIs [1]. OAuth and API keys are typically applied to control access to shared resources [11].

The following subsections deal with two examples related to data sharing, which may be implemented with any external entity, or potentially with the services such as Zapier or IFTTT, which represent popular variants of iPaaS [22], [23].

B. Case scenario context and requirements

Assume that external service periodically checks user's email box, selects and retrieves particular messages, extracts their data, and integrates that data further into its process flow. The user wants to restrict the service to retrieve only emails that are classified under the label *Status updates* and that are no older than one day relative to the temporal point of retrieval.

Following this requirement and Gmail API workflow, the user's security goal can be formulated in a free form as the following:

Allow retrieval of list of messages at user's Gmail account
Allow retrieval of messages classified under 'Status updates' label, considering only the messages arrived today

The first operation, retrieval of list of messages, is enabled by issuing `GET` request to the following API endpoint:

```
/gmail/v1/users/{userId}/messages
```

This endpoint consists of several statically defined elements, with the element `{userId}` corresponding to an *id* of the mailbox owner.

After the client retrieves the list of emails, each of the items in the list will contain basic data about available emails. This data includes email identifier, which can be used to retrieve a particular message with its accompanying details. The API endpoint for the retrieval of single messages is defined by the following skeleton:

```
/gmail/v1/users/{userId}/messages/{id}
```

The issue with executing the workflow of current case scenario, from the perspective of the user and considering the security capabilities of Gmail service, is that only one of the following access scopes can be associated with the client to give the necessary authorization for executing the workflow:

```
https://mail.google.com/
https://www.googleapis.com/auth/gmail.modify
https://www.googleapis.com/auth/gmail.readonly
https://www.googleapis.com/auth/gmail.metadata
```

According to Gmail REST API [24], the first two scopes provide broad access to API functionality, which also includes the account-wide modification and deletion of resources and settings. From the items present on this list, the third scope covering read-only functions could be considered as partially narrow for the current case scenario. However, this scope enables clients to read all Gmail account resources and metadata [25], which among others include, a full history of mailbox changes, the content of all mailbox messages, and their attachments, the content of all currently active message drafts, and user's settings.

Considering that the scopes designed under Gmail's OAuth 2.0 implementation are not appropriate for the security goal expressed above, the following sections exhibit and discuss the policies that allow the conformance to the security goal.

C. Retrieve a list of emails

Figure 7 depicts the fragment of the security policy targeting the first workflow step concerned with the retrieval of the list of email messages available in the user's account.

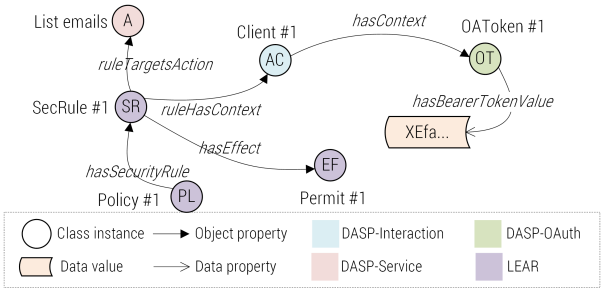

Fig. 7. Security policy to allow providing a list of emails

Referring back to Section III-C with layers of domain-specific vocabularies, we can observe the reliance on particular vocabularies from the DASP framework in the presented security policy. Circles on the figure represent *instances* of classes defined in the vocabulary that is represented by the *color* of the circle. The instances serve as *A-Box knowledge* [16] that renders information on API resources and their structure, policies and other objects. This arrangement is based on the shared conceptualizations established through the T-Box knowledge given in each domain-specific vocabulary [26].

Policy #1 in Figure 7 is an instantiation of the class *SecurityPolicy* from *LEAR* vocabulary. It contains one security

rule *SecRule #1* whose context has to evaluate to true in order for security effect represented by the class of instance *Permit #1* to hold. In a default configuration, if no context, rule or policy is found to hold and set a positive evaluation, the system considers the request as not allowed.

This policy refers to one particular context and one action. The action *List emails* is an instance of a class *Action* from *DASP-Service* vocabulary. Actions are exposed by a service model, which is retrieved during the policy specification phase and whose instances are referenced from the policy. For readability reasons, the service model and its relationships are not included in the figure, and hence *List emails* appears to be isolated in the policy specification.

The context *Client #1* represents a class of *AccessClient* from *DASP-Interaction* vocabulary. In order for the underlying conditions represented by this instance to hold, all its contexts also have to hold. In this case, the client that connects to service has to provide OAuth 2.0 bearer token represented by *OAToken #1* and its value, as shown on the figure. Note that here other conditions can be used or added to this particular client. For instance, by relying on properties defined under *DASP-Interaction* vocabulary, the client could be restricted to a source IP address or a geographical region as well.

Once the contextual condition for the first security rule is met, the rule is allowed to evaluate the positive result represented with *Permit #1* effect. The evaluating environment then interprets given an evaluation and allows the intercepted transaction.

D. Retrieve single email satisfying particular criteria

The API endpoint for the retrieval of single messages answers on `GET` calls by delivering the structured representation of the requested resource. Figure 8 shows the excerpt of that representation using a JSON message format.

The second part of the previously stated user security goal relates to two entities in the message structure, shown in the figure under `labelIds` and `internalDate` fields. In deciding on client request, the policy enforcement system has to check if the corresponding fields of the structured resource conform to requirements given in the authoritative security policy.


```
{  "id": "16b31ea083f8c77d","threadId": "16b31dece18f8f71",
   "labelIds": [ "IMPORTANT", "STATUS_UPDATES", "INBOX" ],
   "snippet": "An amount of EUR 100,00 with the following description ...",
   "historyId": "14271423",  "internalDate": "1559910545000",
   "payload": { ... }, "sizeEstimate": 7993  }
```


Fig. 8. Representation of a single message in Gmail (excerpt)

For this activity, the enforcement system has first to derive the model of a resource structure and then to retrieve and compare values of respective fields with policy requirements. If the requirements in the policy refer to dynamic parameters, the enforcement system has to resolve the corresponding values in order to be able to reach the policy decision.

Figure 9 shows the example of policy that addresses the requirement of the second workflow step. *Policy #2* is an instance of class *SecurityPolicy* from *LEAR* vocabulary. It contains one security rule *SecRule #2* whose contexts have

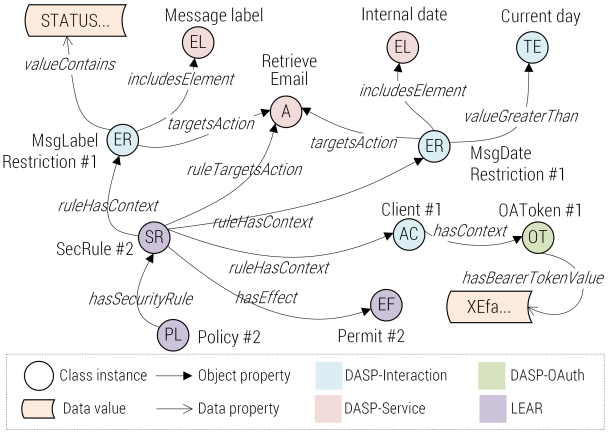

Fig. 9. Security policy to restrict retrieval of single emails

to evaluate to true in order for security effect represented by the class of instance *Permit #2* to hold.

In this particular example, we identify three contextual conditions. First, *Client #1* represents a class of *AccessClient* from *DASP-Interaction* vocabulary. In order for the underlying conditions represented by this instance to hold, all its contexts also have to hold. In this case, the client that connects to service has to provide OAuth 2.0 bearer token represented by *OAToken #1* and its value, as shown in the figure.

The second and third contextual conditions are expressed by nodes *MsgLabel Restriction #1* and *MsgDate Restriction #1*. These objects are instances of class *ElementRestriction* from *DASP-Interaction* vocabulary. The purpose of this class, defined under the hierarchy of *Intrinsic Context*, is to express a state that refers to the entity present in the structure of the target asset. The first restriction refers to the output of *RetrieveEmail* action, where the output element related to *Message Label* has to conform to the rule expressed by data property in *MsgLabel Restriction #1*. In this specific case, the field corresponding to the message label, as identified and classified by Gmail, has to contain the value related to *status updates*. Should the element value fail to deliver a match, the restriction could not hold and the permissive rule, as well as its corresponding policy, would not be applicable.

Similarly, the third contextual condition imposes the restriction for the subjected email message to be received during the *current day*. *Current day* is the instance of the class *TodayEpoch*, defined as a part of the hierarchy of *Extrinsic Context* in *DASP-Interaction*.

It should be noted that Figure 9 provides a simplified policy view for space reasons and clarity. Some nodes from the figure exhibit additional relationships with other parts of the model. The depiction of these entities is not necessary for the illustration of the policy specification mechanism.

Note also the labeling of instances in the figures provided above. As these representations exist in the same unified security model stored in a Δ_{GW} system related to a service and a tenant, a particular degree of concept reuse is applicable and necessary. For instance, *Policy #1* and *Policy #2* are different instances of the same concept; they hold different

identifiers and refer to different security goals. However, *Client #1* and *OAToken #1* from both figures represent the same instances in the unified model.

This reuse of concepts shows that various distinct policies and rules may reference a single registered client, if necessary. The same approach applies to actions exposed by GMail Web API specification, which is also part of a unified security model stored in the gateway.

E. Summary

The examples provided in this section illustrate the specification of two security policies implementing the user security goal using gradual levels of complexity and expressivity. The first policy illustrates the basic underlying idea behind the policy specification, which is the association of security rules with client actions (intents) and the reliance on the evaluation of rule contexts to allow the policy decision to be actuated in the form of a corresponding policy effect.

In the first example, we can observe the reliance on instances (nodes) from several vocabularies. The specification of an API is exposed by a service using *DASP-Service* vocabulary and integrated into a unified security model utilized by the gateway instance to analyze requests and enforce policies. The policies specified by the resource owner rely on this API description by referencing its entities in the policy designation process executed *a priori* to client accesses. As *LEAR* policy vocabulary acts agnostically in terms of a mechanism employed to authenticate the accessing client, the supplementary nodes shown in the example express a particular authentication mechanism represented by their source vocabularies.

As a result of integrating the specifications drawn by the service and the user, Δ_{gw} derives a unified model consisting of interrelated API and policy specifications. This model gets eventually enhanced and updated at the time of client accesses to extend the horizontal plane of contextual conditions by describing requests occurring in the domain.

While the practical implementation of the system may vary in terms of the application of these capabilities, the enhanced version of the unified model allows the interweaving of both *vertical* and *horizontal* specification, connecting both structural and temporal dimensions of the client accesses. This further allows the expression of cross-request contextual conditions, such as defining a sequence of allowed interactions or specifying the allowable number of client requests per time slice.

The second example described in this section illustrates the application of contextual restrictions, which control the access to API resources based on environmental states. These states can express the *intrinsic* and *extrinsic* conditions, which are then used to decide on the applicability of security rules. The reliance on contextual restrictions allows the specification of fine-grained security requirements that allow integration of dynamic and transparent evaluation properties related to a resource, process, or external environment.

VI. Evaluation and Discussion

A. Authorization middleware

We have implemented Δ_{GW} prototype as a modular application based on Java, PlayFramework and OWL-API. Our design aims at providing containerized, scalable instances that serve resources from a single-tenant domain, and that can be easily extended with additional functionality according to the client needs.

1) Test configuration: To investigate the practical use of our prototype, we have focused on two important phases in the execution of Δ_{GW} interception and security evaluation functionality. The first phase corresponds to the identification of policies relevant for a particular intent derived from the intercepted request. After their identification, security policies are subjected to the second phase of policy evaluation. This activity comprises the retrieval of data referenced in policy and the evaluation of all applicable contextual conditions. Based on this evaluation, the system decides whether the request is applicable or not, and which effects (obligations) have to be imposed.

Our test configuration includes four sets T1-T4 of registered policies for the protected API, ranging from 1 (T1), through 50 (T2) to 250 policies (T3 and T4). All the policies from sets T1, T2 and T4 are applicable for a given test request. Contrary to that, in set T3 only 40% of policies are applicable for the given intent and thus subjected to evaluation.

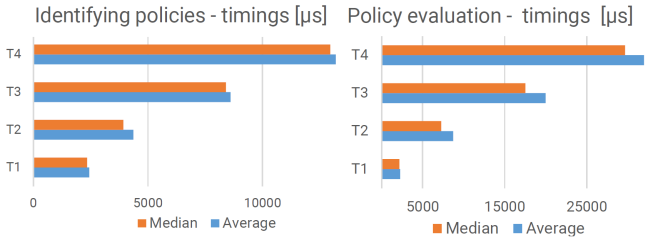

Fig. 10. Identification and evaluation of relevant security policies

2) Results: We have executed each of the said phases on Amazon EC2 instance of *c5d.2xlarge* type with 8 vCPU and 16 GB RAM, running Amazon Linux 2 x86_64 on Intel 8124M CPU. We gathered more than 100 subsequent data observations. Figure 10 presents observed results.

Our results indicate linear dependency between evaluation timings and the size of the policy store. Most tenants tend to deploy less than 50 policies, which renders expected policy evaluation in an average case to take up to 10 ms per client access. Our goal for the next prototype iteration is to reduce overhead at 1 ms per request.

B. Synergy with other frameworks

1) Service description frameworks: OpenAPI [27] is the effort organized and backed by Linux Foundation and several industry players. It builds on Swagger, an initiative aimed at developing an open specification for machine-readable interfaces for describing, producing, consuming, and visualizing RESTful services. OpenAPI is widely used by service

providers in the process of service development, testing, and documenting. At the end of the year 2019 popular directory, *APIs.guru* provides more than 500 regularly updated specifications of public APIs.

A service provider wishing to use DASP components can reuse its existing OpenAPI descriptions in building a service model, which can be generated by importing *resources*, *operations* and *data structures* from OpenAPI interface description. To become operational, the resulting model needs to be extended with additional properties given in DASP vocabularies. This approach allows the rapid definition of new service models by relying on the existing assets.

2) Authentication frameworks: DASP integrates support for credential-based authentication mechanisms such as *bearer tokens* in OAuth 2 [12] or *API-keys* [28]. The use of bearer tokens for clients is illustrated in the example from Section V-D and on Figure 9. This integration is transparent for the client as Δ_{gw} can be involved in authorization flows and issuance of tokens. Furthermore, internal representations of tokens in Δ_{gw} can be synchronized with or dynamicaly managed based on external, authoritative repositories.

The support for JWT [29], [30] as more advanced signature-based authentication means is provided using separate vocabulary as well. Similarly, as in the case of bearer tokens, resources for verification of JWT signatures can be provided in an integrated model or synchronized with the external store. DASP can be extended to support other signature-based approaches by defining a vocabulary for a particular approach and implementing its related software module.

As DASP relies on an abstract concept of a *client*, which can be characterized by one or more contexts (as shown in Figure 9), authentication mechanisms based on other frameworks can be combined with the other supported contextual parameters. This way, the use of a particular external mechanism can be restricted based on geo-region, IP network, access count, request sequence, or time-of-day, for instance.

C. Limitations and future work

Our solution currently focuses on Web API related interactions. It should be noted that other mechanisms exist to support resource sharing across entities, such as *WebSockets*, *WebRTC*, or various messaging-based solutions [31]. While the structure of layers, as shown in Fig. 1 allows the introduction of new techniques, their definition, and implementation in practice require additional research efforts.

A growing degree of data, data sharing, and numerous interconnections per se impose additional cognitive overhead in managing the security of these interactions. From that view, we envisage two further research directions to address this issue. The first branch relates to the analysis of usability aspects and the development of assistive user interfaces to simplify authorization management. The second direction relates to the use of autonomous agents. As one of the goals of our contribution was to render authorization primitives machine-understandable, the next step would be further research to enable agents to independently analyze and suggest actions related to personal and organizational data and sharing contracts at and among diverse providers.

Finally, in terms of technical development, we intend to optimize the performance of our implementation and provide modules that would allow its direct integration with popular gateways such as *Nginx*.

VII. Related Work

Several initiatives emerged to facilitate interoperability and connectivity by the means of open vocabularies. These include Hydra [32], for the creation of generic API clients, and LOV4IoT for the integration of IoT devices [33]. LOV by Vandenbussche et al. [34] is envisaged to support the overall consolidation of linked vocabularies . Our work is complementary to these approaches as it aims to fill the gap for specifications in the security domain.

Hüffmeyer and Schreier proposed RestACL language for the protection of RESTful services [35]. Alam et al. [36] developed xDAuth framework that supports authorization and delegation in the RESTful service architecture. Beer et al. considered the architectural aspects of protecting access to RESTful service [37]. Both of these approaches consider the perspective of a single enterprise, providing the access specification capabilities on a granularity level of a resource, without tackling contextual dynamics

SUNFISH [38] is an EU-supported initiative to develop a framework for the establishment of secure cloud-based service federations. SUNFISH externalizes authorization beyond a single organization by providing consolidated policy model and out-of- the-box support for dynamic resource transformation. It, however, focuses on traditional web services and imposes high adoption barriers for entities beyond public administrations.

VIII. Conclusion

In this work, we introduced the framework that supports transparent externalization of authorization functionality for access control of Web API resources. In our approach, we leverage the notion of *knowledge graph* and apply an integral, model-based view that combines concepts from domains of service descriptions, data formats, security policies, and service interactions. Envisaged as a dynamic database that integrates structural and temporal descriptions of service interactions, our proposal allows feature-rich specification and enforcement of user security goals over its resources hosted at distributed services.

In contrast to other approaches, the presented framework allows security policies to evaluate structural elements of protected resources and perform an online transformation of shared data using diverse intrinsic and extrinsic contextual properties. Our solution supports service providers in reducing efforts to implement comprehensive access control functionality for the sharing of tenant resources. In parallel, tenants benefit from the fine-grained security capabilities, on-premise enforcement, and the consolidated reuse of the same framework across different services.

REFERENCES

[1] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public rest web service apis," *IEEE Transactions on Services Computing*, 2018.

[2] D. Guinard and V. Trifa, "Building the web of things," *Shelter Island: Manning*, 2016.

[3] M. A. Oumaziz, A. Belkhir, T. Vacher, E. Beaudry, X. Blanc, J.-R. Falleri, and N. Moha, "Empirical study on rest apis usage in android mobile applications," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 614–622.

[4] A. Jøsang, "A consistent definition of authorization," in *International Workshop on Security and Trust Management*. Springer, 2017, pp. 134–144.

[5] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[6] O. X. T. Committee *et al.*, "eXtensible Access Control Markup Language (XACML) Version 3.0," *Oasis standard, OASIS*, 2013.

[7] T. Hardjono, E. Maler, M. Machulak, and D. Catalano, "User-managed access (UMA) profile of OAuth 2.0," *Internet Engineering Task Force (IETF)*, 2015.

[8] M. P. Machulak, E. L. Maler, D. Catalano, and A. Van Moorsel, "User-managed access to web resources," in *Proceedings of the 6th ACM workshop on Digital identity management*. ACM, 2010, pp. 35–44.

[9] O. for Economic Co-operation and Development, *Data-Driven Innovation: Big Data for Growth and Well-Being*, 2015.

[10] F. Bülthoff and M. Maleshkova, "RESTful or RESTless–Current state of today's top Web APIs," in *European Semantic Web Conference*. Springer, 2014, pp. 64–74.

[11] B. Suzic, B. Prünster, and D. Ziegler, "On the structure and authorization management of restful web services," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 2018, pp. 1716–1724.

[12] M. Jones and D. Hardt, "The oauth 2.0 authorization framework: Bearer token usage," Tech. Rep., 2012.

[13] P. Siriwardena, "Oauth 2.0 profiles," in *Advanced API Security*. Springer, 2020, pp. 211–226.

[14] H. van der Veer and A. Wiles, "Achieving technical interoperability," *European Telecommunications Standards Institute*, 2008.

[15] I. Salvadori and F. Siqueira, "A maturity model for semantic restful web apis," in *Web Services (ICWS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 703–710.

[16] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: principles and methods," *Data & knowledge engineering*, vol. 25, no. 1-2, pp. 161–197, 1998.

[17] R. Verborgh, E. Mannnens, and R. Van de Walle, "Bottom-up web apis with self-descriptive responses," in *Proceedings of the First Karlsruhe Service Summit Workshop-Advances in Service Research*, 2015, p. 143.

[18] A-SIT, "Dasp - data sharing and processing framework," http://daspsec.org, 2018.

[19] F. B. Schneider, "Least privilege and more [computer security]," *IEEE Security & Privacy*, vol. 99, no. 5, pp. 55–59, 2003.

[20] S. Goessner, "JSONPath - XPath for JSON," 2007, retrieved from: http://goessner.net/articles/JsonPath/.

[21] R. T. Fielding, R. N. Taylor, J. R. Erenkrantz, M. M. Gorlick, J. Whitehead, R. Khare, and P. Oreizy, "Reflections on the rest architectural style and principled design of the modern web architecture (impact paper award)," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 4–14.

[22] M. Pezzini, "Integration paas: enabling the global integrated enterprise," *Gartner Application Architecture, Development & Integration Summit, Las Vegas*, vol. 29, 2011.

[23] M. Marian, "ipaas: Different ways of thinking," *Procedia Economics and Finance*, vol. 3, pp. 1093–1098, 2012.

[24] Google, "Users.messages: get," https://developers.google.com/gmail/api/v1/reference/users/messages/get, 2019.

[25] ——, "Choose auth scopes," https://developers.google.com/gmail/api/auth/scopes, 2019.

[26] M. Bauer, H. Baqa, S. Bilbao, A. Corchero, L. Daniele, I. Esnaola, I. Fernández, Ö. Frånberg, R. G. Castro, M. Girod-Genet *et al.*, "Semantic iot solutions-a developer perspective," 2019.

[27] OpenAPI Initiative, "The OpenAPI Specification," 2018, retrieved from: https://github.com/OAI/OpenAPI-Specification.

[28] S. Farrell, "Api keys to the kingdom," *IEEE Internet Computing*, vol. 13, no. 5, 2009.

[29] M. B. Jones, "The emerging json-based identity protocol suite," in *W3C workshop on identity in the browser*, 2011, pp. 1–3.

[30] J. Bradley, N. Sakimura, and M. B. Jones, "Json web token (jwt)," 2015.

[31] P. Krawiec, M. Sosnowski, J. M. Batalla, C. X. Mavromoustakis, G. Mastorakis, and E. Pallis, "Survey on technologies for enabling real-time communication in the web of things," in *Beyond the Internet of Things*. Springer, 2017, pp. 323–339.

[32] M. Lanthaler and C. Gütl, "Hydra: A vocabulary for hypermedia-driven web apis." *LDOW*, vol. 996, 2013.

[33] A. Gyrard, G. Atemezing, C. Bonnet, K. Boudaoud, and M. Serrano, "Reusing and unifying background knowledge for internet of things with lov4iot," in *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on*. IEEE, 2016, pp. 262–269.

[34] P.-Y. Vandenbussche, G. A. Atemezing, M. Poveda-Villalón, and B. Vatant, "Linked open vocabularies (lov): a gateway to reusable semantic vocabularies on the web," *Semantic Web*, vol. 8, no. 3, pp. 437–452, 2017.

[35] M. Hüffmeyer and U. Schreier, "Restacl: An access control language for restful services," in *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*. ACM, 2016, pp. 58–67.

[36] M. Alam, X. Zhang, K. Khan, and G. Ali, "xdauth: a scalable and lightweight framework for cross domain access control and delegation," in *Proceedings of the 16th ACM symposium on Access control models and technologies*. ACM, 2011, pp. 31–40.

[37] M. I. Beer and M. F. Hassan, "Adaptive security architecture for protecting restful web services in enterprise computing environment," *Service Oriented Computing and Applications*, vol. 12, no. 2, pp. 111–121, 2018.

[38] F. P. Schiavo, V. Sassone, L. Nicoletti, and A. Margheri, "Faas: Federation-as-a-service," *arXiv preprint arXiv:1612.03937*, 2016.