**Zentrum für sichere Informationstechnologie – Austria**
**Secure Information Technology Center – Austria**

A-1030 Wien, Seidlgasse 22 / 9
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

A-8010 Graz, Inffeldgasse 16a
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

# AUTOMATED REASONING OVER SECURITY POLICIES

*Project Report*

*Version 1.0, 20.4.2017*

*Bojan Suzic – bojan.suzic@a-sit.at*

***Abstract:***

Applied approaches on authorization management often focus on a single system or environment, neglecting the need to address the security of the data sharing processes that span various entities and organizations. Several consequences to security management in cross-organizational processes can be observed: First, cloud providers are not capable of offering their users a consolidated and cross-provider interoperable interface or approach to manage the security controls applied to their resources. Instead, providers usually expose provider-specific and service-specific interfaces with reduced functionality, imposing the management and integration overhead to users that consume many cloud resources. Second, existing multilateral authorization management approaches, such as the OAuth 2 framework, address the security capabilities of protected resources only partially. The design approach behind this framework imposes the static and coarse-grained means to perform authorization capabilities, which are usually conforming only to the requirements and perspective of the cloud provider.
Approaches such as XACML enable rich definition of authorization policies; they are however coupled to particular enterprises, their processes and service models, missing to provide standardized means to control or automate cross-entity interactions.

In the course of this work, we address the shortcomings of existing frameworks by separating authorization management from particular organizations, their business or resource models. We establish a framework that defines abstract means to manage the security of resources distributed across diverse services using a unified service and policy description models. Our proposal relies on approaches from semantic technology stack, including RDFS, OWL, and SWRL standards for knowledge serialization and management. In this work, we particularly focus on the application of reasoning techniques that enable automated inference of facts based on the models and knowledge provided in different ontologies and their practical instantiations. By integrating models and descriptions from several perspectives, we enable management of security controls that is less dependent or related to particular service and focused on needs of users.

This report introduces the general approach, proposed architecture and, by applying the running case-scenario, explains the practical application of the proposed model. The report further presents and discusses the initial evaluation of performances of proposed approach.

## *Contents*

## Figures

## *Tables*

# 1. Introduction

The overall cloudification of services and an increasing amount of data that is stored in, delivered from or exchanged between different systems deployed in the cloud requires rethinking the existing paradigms of security and privacy management. The use-cases enclosing the traditional environment and architectural models assumed the user's or organizational data to be primarily stored and processed at a single entity, inside the premises of a particular organization. In contrast to that, the business models currently in a broader application, as well as their emerging derivations, assume the dynamic processing and reuse of data in multi-sectoral transactions. This expectation also implies the sharing and processing of data between different organizational entities, for various purposes and beyond its original or primary motivation.

The requirement for proper management of security and privacy of data consumed in present dynamic and distributed environments adds overhead to the complete process of data exchange and reuse. Several important points should be observed here.
First, the currently applied solutions deliver suboptimal scalability of control. Most of the protocols or architectures focus on the data exchange or reuse across two entities, in a closed environment. Their models introduce the coupling, both on the level of data representations, as well as on the level of services, interfaces and organizational processes. The resulting degree of coupling further introduces the unnecessary administrative overhead in the phases of development, integration, and maintenance of the services and especially their integrations. By imposing the overhead for interoperability in one-to-many or many-to-many interaction scenarios, this coupling effectively hinders the broader adoption of existing scenarios or the emergence of novel business models.

Secondly, the complexity of interactions and interconnections at degree present in today's typical setting is not trivial to represent and manage using existing approaches, which were originally aimed for less complex scenarios. The primary question here is how an average user can efficiently manage its data present on different services? How can a user control the processing and exchange of its data among various cloud services? Is it possible to achieve such control by additionally considering contextual awareness and dynamic restrictions?

While the management of security and privacy in closed or restricted environments[1] can be easily attained by applying the approaches available since decades, the scenario scaled to tens or hundreds of different providers, real-time data exchange and the existence of many different datasets is significantly more challenging. From the perspective of the user, this challenge is reflected through the cognitive and time-related overload resulting from different tools, processes, and rules that need to be separately executed and periodically reviewed for each separate platform. The hosting organization has, on the other side, to design, implement and expose security-related controls for each service using its available resources. It also has to provide the integration with other services as an ongoing process in order to foster external innovation and stay competitive in the business world. Moreover, the developers have to learn, integrate and maintain each of these systems and integrations.

Recent protocols, such as OAuth [1] and UMA, address the challenge of cross-organizational authorization from a simplified view. The detailed evaluation of these solutions has been provided in [2, 3]. The XACML architectural and protocol model delivers complex solution aimed for enterprise-related use cases and primarily intended for closed environments. Still, it is highly coupled in restricted in flexibility.

Our work addresses the complexity of security management in many-to-many interactions by providing the reusable modeling framework and tools for multilateral security management. By relying on techniques of semantic modeling and rule-based inference, the proposed approach reduces the coupling between data modeling and implementations, reducing the interoperability overhead, enabling novel flexible applications and introducing forward-compatibility for different

[1] Such as the interaction environments consisting of two or several parties

scenarios and domains. The additional application of ontology and rule-based reasoning in this work enables the automated and efficient management of complex descriptions and flows by relying on separate and extensible constraints and rules integrated into the model.

## 2. Architectural Model

*In the scope of this work, we have developed an experimental framework to evaluate and test interoperable, integrated and automated security management approach.*

Figure 1 presents the overall architecture of the framework. The subsequent subsections provide the general overview and discuss its comprising components.

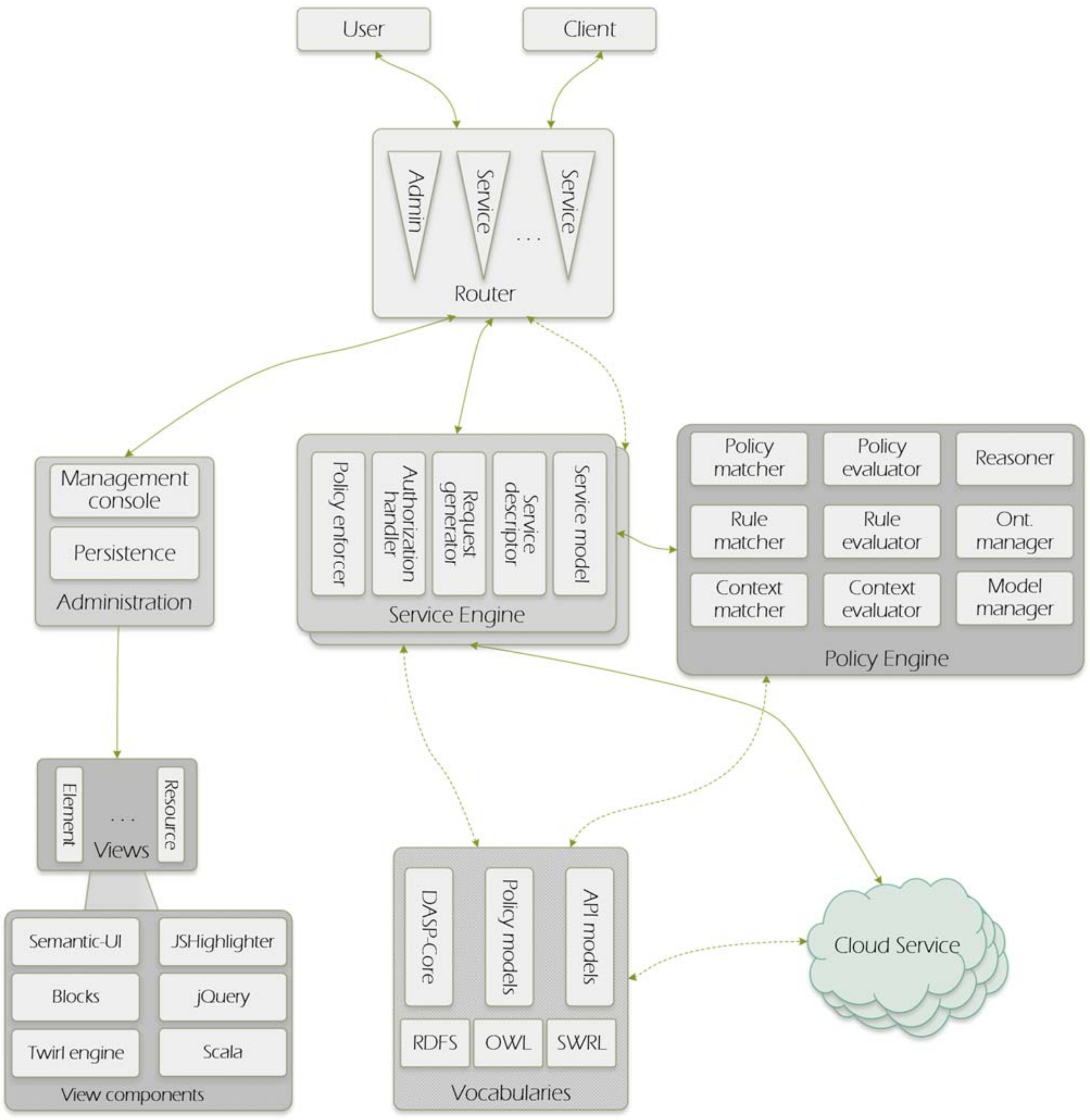


*Figure 1: Overview of the architecture*

## 2.1. General overview

The prototypic implementation of the proposed framework is a service middleware that operates between a user[2] or a client[3] and the target cloud service that hosts the resources. The purpose of this system is to enable the user to apply advanced security management capabilities and control the usage or processing of its resources deployed at various external third-party cloud services. Therefore, the framework provides two primary interfaces for requestors. The first one is intended for the user, enabling it to centrally and integrally manage the security of its resources dispersed over various cloud providers. The second interface aims to serve the clients that request these resources by acting as a transparent security gateway sitting between a client and requested cloud service.

## 2.2. Service engine

The proposed framework is intended to support various cloud services. For this purpose, an abstract component that implements a general functionality of a service engine is provided, allowing the implementing components to reuse the common functionality and integrate support for a service-specific API. The service engine defines a *Service model*, which offers the model of a supported service using the common vocabulary. *Service descriptor* provides additional data and functions of a supported service that are relevant for clients and backend applications.

As it acts as a gateway between a client and the service, on each client request, the *service engine* generates a *request model* using a common vocabulary. This model is populated with the contextual properties of a client, request, its environment and the target resource. In the further phases of a request-response cycle, the *service engine* delegates the authorization to service-specific authorization handler, which is primarily applied to authorize the middleware against the target cloud service. Both in the initial and in the final phase of request-response cycle, the *service engine* reuses the functionality of the *policy engine* to initiate a policy-based request and response evaluation. The results of these processes are enforced by the *policy enforcer*, the component that dynamically transforms request and responses to user's policies.

## 2.3. Policy engine

The policy engine is a component employed to evaluate both the requests from clients and responses from the services. From this point, this component checks the conformance of each interaction cycle against user's policies, allowing the execution of corrective or prohibitive measures, as defined in the policies. For this purpose, it relies on a *service (API) model*, *request model* and *user's policies*, which are previously established using common and abstract vocabulary. In later steps and at each interaction these models are correlated and applied to derive additional knowledge that is necessary to evaluate the interaction.

Due to the complexity of service models and user policies, it is not suitable to define each possible relation and interdependence between particular resource instances, policies and requests. This especially applies for each potential combination of contextual or environmental parameters. Instead, the proposed framework relies on overall and abstract goals that are on-the-fly translated to specific authorization decisions. The component that enables this transition is the *reasoner*, which, based on a general set of rules, reasons over provided models and infers the information and knowledge necessary to evaluate particular interactions.
This approach enables modular contextual properties to be injected in the models at run-time, allowing the evaluation of contextual parameters on various levels of granularity.

Besides reasoning functionality, policy engine provides the interfaces for deriving relevant *policies*, *rules*, and *contexts* for each request or interaction. These are used by service engine and other components in the process of establishing and enhancing the descriptions of request model.

## 2.4. Vocabularies

This component provides the vocabularies reused in the previously described components. Principally, the particular instances of *requests*, *API models,* and *policy models* are derived using common vocabularies available in this component. The first range of vocabularies is *DASP-Core [2, 4]*, which establishes the core concepts and relations that are instantiated in other vocabularies. *Policy models* are the instances of

[2] In the terminology of this work, a user is considered as a resource owner that hosts its resources at some cloud service. It also can be a subscriber of a cloud service.

[3] In the scope of this work, a client is an automated agent or other entity that tries to access the resources of a user, hosted at or offered by third-party cloud service.

particular service-specific policies created by users. They rely on *API models*, which provide structural descriptions of individual cloud services using common resource classes, object and data properties. The final instance of resources provided in this component are the *rule transitions*, which establish the rules applied to infer the knowledge and connect the instances in the models.

There are three primary building blocks applied in the definition of vocabularies. OWL *[5]* is a primary mean for establishing and instantiating vocabularies and specifying the semantics of the schemas. The resources and resource hierarchies are structured using OWL *classes* and *instances*, with the *object properties* describing the relations between different resource instances and *data properties* providing the data values. These means are complemented with other OWL capabilities. *Domains* and *ranges* are used to establish the restrictions on object properties, while *equality* and *inequality* are applied to relate and distinguish between instances of resources in the model. The resources and properties are hierarchically ordered to allow different levels of abstraction in the practical integration.

Resource Description Framework Schema (RDFS) *[6]* is the second technology used to describe the resources and policies. Being less expressive as Web Ontology Language (OWL), RDFS is applied mostly to provide serializations of the models or to define simple models. In contrast to that, Semantic Web Rule Language (SWRL) *[7]* is employed to define general rules applied in the process of the reasoning. These rules are executed in the reasoner component, allowing to achieve the conclusions about the type of the relations and properties between the instances in the model.

# 3. Discussion

In this chapter, we explain the functionality of the implemented model by considering a restricted running example. We also present the initial performance tests and discuss the results.

## 3.1. Practical Example

This section presents some examples applied on a use-case with the purpose to illustrate the application of the framework and its underlying approach. For this purpose, we first illustrate an overall processing approach, as presented in *Figure 2*.

In this process, we first produce *API Model*, which describes a Web API of a particular service. Then, the user instantiates the *Security Policy Model*, which defines a range of allowed operations and dynamic transformations applied to the resource representations under particular contextual conditions. Policies reuse the API descriptions exposed in the *API Model* and potentially refer to particular properties of API resources or clients. These properties may include i.e. authorization token, IP address, source network or a similar property of the client, or a resource *id*, resource or element *value* or an *abstract property*, as exposed by the API. During the access phase, each request is modeled using available data from the request (such as client's properties) and API model. Based on these data, predefined SWRL rules, OWL restrictions, and capabilities presented in the API Model, the reasoner infers additional knowledge about the request and enhances all models with supplementary statements.

This approach hence enables the definition of rich API models and relationships between its elements as well as the abstract and flexible definition of policies without the need to handle each possible combination of resources, access policies, and effective requests. It also enables the consolidated definition of policies that deal with the resources exposed under different clouds and APIs, at once.

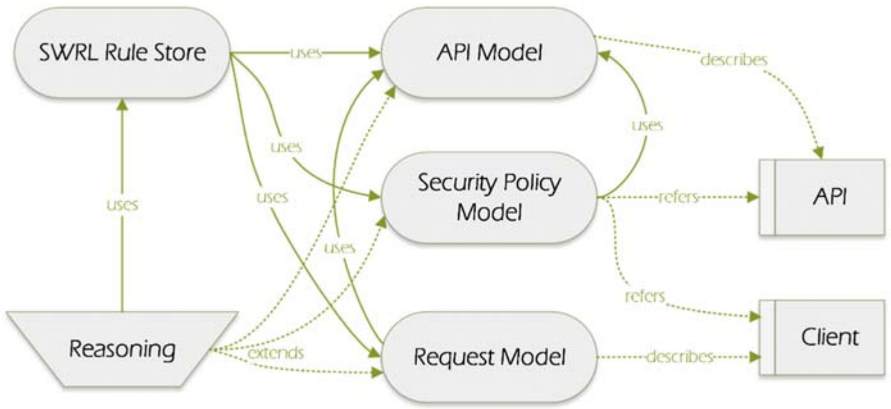


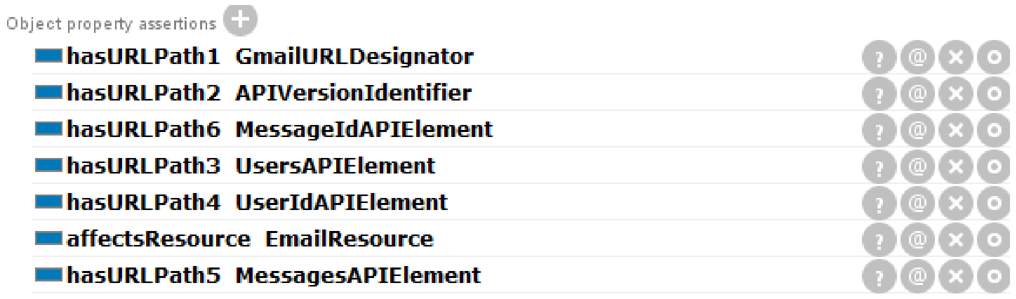
*Figure 2: Enforcing the policies*

To illustrate the approach using a running example, let's say that Gmail is a *service* that exposes *email resource*, and particularly, an *action to retrieve* the email. This action is characterized by firing an *HTTP GET* request to the *path with the specific pattern* consisting both of *fixed* and *variable elements*, of whom *some* serve as *identifiers* of the target resource. The resource itself has a complex structure that is *contextually sensitive*[4] and in its default instance consist of several *elements*. Each of these elements has a specific *structure,* which may be expressed with JSON or XML patterns.

This is a complex statement that describes only a portion of the overall Gmail service's API and its functionality. This and all other statements that describe APIs, requests or policies are structured using DASP-Core Vocabulary *[2, 4]*, which establishes core concepts, relationships, and restrictions to support consolidated and formalized descriptions that are reusable and extendable across different systems.

*Figure 3* provides the graphical description of a part of that statement that relates to the action of retrieving an email[5]. Each element in that description represents an instance of a class defined by an OWL object property range. Hence, the elements under *hasURLPath** property under DASP-Core vocabulary are expected to be the members of *FixedPathElement* or *DynamicPathElement* classes. While the first one establishes the URL path elements with fixed path elements, the second one provides the means to express the dynamic elements in API call path.

The related example in the case of Gmail service corresponds to the following path: `/gmail/v1/users/<userid>/messages/<id>`, with dynamic elements appearing in brackets. These elements can be, for instance, used in policies to identify a particular range of resources or a resource with the specific property and handle their integration with additional measures.



| | |
|---|---|
| hasURLPath1 | GmailURLDesignator |
| hasURLPath2 | APIVersionIdentifier |
| hasURLPath6 | MessageIdAPIElement |
| hasURLPath3 | UsersAPIElement |
| hasURLPath4 | UserIdAPIElement |
| affectsResource | EmailResource |
| hasURLPath5 | MessagesAPIElement |

*Figure 3: Sample action description (retrieve email)*

The action description, as provided in the previous figure, is used to find relevant policies for a request conforming a particular URI pattern that satisfies additional requirements, such as request type, parameters, or target resource properties. One of the steps in finding these resources, as illustrated in *Figure 2*, is the process of reasoning. Based on 1) constraints and relations provided in OWL model, and 2) SWRL rules defined for additional inference, the reasoner derives additional facts about the models and enhances these with detailed descriptions. The example of that process is provided on Figure 4, which describes the enhanced request model after the reasoning is completed.

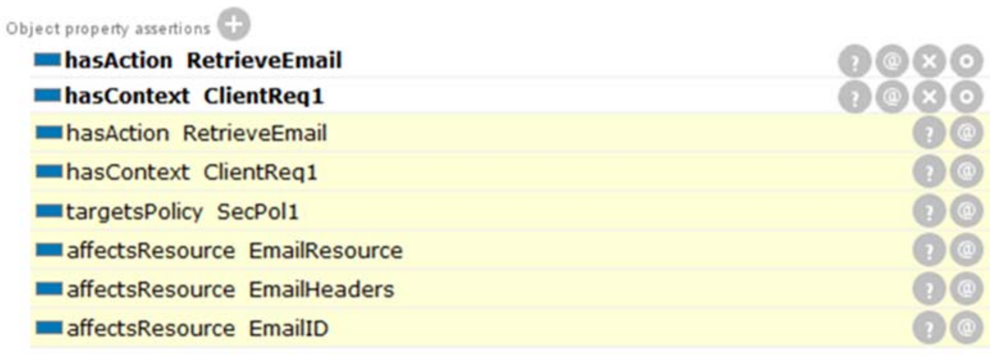
The first two statements on the figure are established in the initial phase of request and policy matching. In this step, the system finds matching action(s) and describes the client by reusing a known vocabulary. In the second phase, based on the predefined policies, request parameters and relationships between the elements in the model, the reasoner infers additional facts (yellow background on the figure) that help to find matching policies, resources and their elements affected by request.

In the particular example, the reasoner finds that the email headers and email id, which are part of an email resource, are indirectly affected by request (as they are exposed by default). It then finds the relevant policy that refers to one of these resources. The basis for the inference is established by relationships expressed with OWL and by explicit rules provided in SWRL. While the OWL relationships are an inherent part of

[4] The particular representation of exposed resource (and the related inclusion of other elements) depends on a combination of input parameters in the request. This way, for example, specific request input parameters may define the message resource representation to be derived with additional data elements included (or excluded).

[5] The figures in this section that relate to model examples are taken using Protégé ontology management tool and integrated Pellet reasoner. The entities with white background relate to instances defined in the model, while the entities and relations shown with yellow background refer to the relationships derived by the reasoner.

DASP-Core vocabularies, the SWRL rules are defined and imported separately and can be adjusted to serve different purposes or use-cases.



Object property assertions

**hasAction RetrieveEmail**
**hasContext ClientReq1**
hasAction RetrieveEmail
hasContext ClientReq1
targetsPolicy SecPol1
affectsResource EmailResource
affectsResource EmailHeaders
affectsResource EmailID

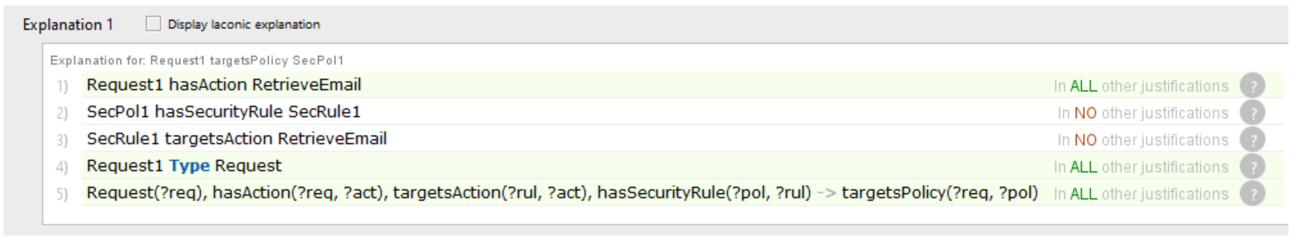
*Figure 4: Sample request description*

Four sample rules provided in *Figure 5* represent an excerpt of rules applied in the framework.

```
Request(?req) ∧ hasAction(?req, ?act) ∧ targetsAction(?rul, ?act) ∧
hasSecurityRule(?pol, ?rul) ⇒ targetsPolicy(?req, ?pol)

Request(?req) ∧ includesParameter(?req, ?param) ∧ hasAction(?req, ?action) ∧
supportsParameter(?action, ?param) ∧ affectsElement(?param, ?resource) ∧
hasSecurityRule(?policy, ?rule) ∧ targetsResource(?rule, ?resource) ⇒
targetsPolicy(?req, ?policy)

Action(?act) ∧ affectsResource(?act, ?res) ∧ hasElement(?res, ?elem) ∧
Element(?elem) ⇒ affectsResource(?act, ?elem)

Request(?req) ∧ Action(?act) ∧ hasAction(?req, ?act) ∧ affectsResource(?act,
?res) ∧ Resource(?res) ⇒ affectsResource(?req, ?res)
```

*Figure 5: Excerpt from SWRL rules applied in the framework*

By relying on such approach, we may define policies on an abstract level, such as on the level of resources that cross different systems or services. We may also define the policies over particular elements, which are exposed as a part of diverse resources. An example of that may be an email header, which could be exposed as a part of different API endpoints, with the appearance in the response depending on request parameters (such as requested detail level or expected fields of the response). Instead to define policies for each possible appearance of email header under different endpoints and circumstances, by relying on dynamic inference we may define one policy for one kind of resource, which is then applied in each interaction and circumstances that may render particular resource.



Explanation 1 ☐ Display laconic explanation

Explanation for: Request1 targetsPolicy SecPol1

1) Request1 hasAction RetrieveEmail — In ALL other justifications
2) SecPol1 hasSecurityRule SecRule1 — In NO other justifications
3) SecRule1 targetsAction RetrieveEmail — In NO other justifications
4) Request1 **Type** Request — In ALL other justifications
5) Request(?req), hasAction(?req, ?act), targetsAction(?rul, ?act), hasSecurityRule(?pol, ?rul) -> targetsPolicy(?req, ?pol) — In ALL other justifications

*Figure 6: Applying reasoning - one of explanations for inference*

The example on *Figure 6* illustrates the inference applied by using one of the SWRL rules presented in *Figure 5*. In this example, the reasoner has, through the application of this rule (line 5), identified a security rule and security policy defined in the model and extended the request to target related policy, as shown using *targetsPolicy* property in Figure 4.

### 3.2. Performance and timings

In the scope of the implementation of the proposed architecture, we have performed initial tests related to the performance and timings of the processing model. The evaluation has been done using single desktop Intel i5 CPU with 8 GB RAM and Java 8. The underlying model relies on experimental Openllet API 2.6.0 which integrates OWL-API 5.x and Jena API 3.x with Pellet reasoner, providing a unified interface to consume functionalities from both libraries.

The performance evaluation has been performed on an experimental implementation without relying on additional optimizations or checks. The intention has been to gather the rough estimation on how much time on a target platform is required to perform different critical phases of the processing. The gathered mean results are provided in *Table 1*.

| Num. | Activity | Timing (ms) |
|---|---|---|
| 1. | Initialize all ontologies | 47 |
| 2. | Instantiating request | 0,07 |
| 3. | Finding relevant actions | 2 |
| 4. | Initialize policy engine | 72 |
| 5. | Initialize reasoner and perform inference | 8 |
| 6. | Policy evaluation | 0,001 |

*Table 1: Excerpt from processing timings*

Some activities illustrated in *Table 1* have to be performed infrequently, such as during the application startup or configuration reload. Such activities include i.e. the initialization of policy engine or initialization of ontologies. The instantiation of the request is related to the processing of each request and it has to be repeatedly performed for separate accesses. The same is true for policy evaluation, which has to be performed with each access twice (for request and response cycles). The reason behind low performance overhead in policy evaluation is the fact that this component is inherent to the framework, locally implemented and it does not significantly rely on external libraries.

In overall, based on the gathered data, in the current configuration and with the sample models, expected overhead added to each request and its evaluation is expected to be in the range of 10 ms (items 2 + 3 + 5 + 6 from *Table 1*). We expect that additional optimizations, such as applying *parallelism* and *reactive* (asymmetric, non-sequential) evaluation, as well as the integration of caching and other smaller optimizations can bring this overhead under 5 ms for APIs and policies of standard complexity.

# 4. Conclusion

This report presents the work performed on integrating reasoning capability in the framework for consolidated and expressive security management of user resources distributed over different cloud services. This framework primarily focuses on the control of resources or services exposed using typical Web API based approaches, such as RESTful interfaces. By separating API models from particular implementations and reusing the integrated framework based on semantic vocabularies, the proposed model enables fine-grained, contextually sensitive, highly expressive and dynamic control of user resources hosted on and exchanged between diverse systems. The reasoning capability extends this framework with the means to perform automated and provable inference of facts, allowing the separation of business logic from implementations and generation of simplified API models and policies without sacrificing expressivity or flexibility.

In this report, we presented the architecture of the framework and, based on a running case, the practical application and relevant processing steps envisaged in the proposed approach. We furthermore present the initial results related to performances of the proposed approach, demonstrating its low overhead and suitability for broader application.

In the further work, we aim to extend the framework with an extensive range of supported APIs, to perform additional performance evaluations and optimizations and to evaluate different architectural scenarios that consider the integration of proposed prototype at various premises. In this sense, we envisage two main deployment scenarios that consider the application both as 1) a user-specific instance that controls the data hosted at diverse clouds and 2) as the cloud service-specific instance that separates authorization management from the cloud provider and exposes advanced and unified security management interface to its end users.

## References

[1] D. Hardt, *The OAuth 2.0 authorization framework.,* 2012.

[2] B. Suzic, *Multidimensional Security Policies,* Graz: Zentrum für sichere Informationstechnologie - Austria (A-SIT), 2016.

[3] B. Suzic, "Securing integration of cloud services in cross-domain distributed environments," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016.

[4] B. Suzic, "User-centered Security Management of API-based Data Integration Workflows," *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium,* pp. 1233-1238, 2016.

[5] W3C Owl Working Group, *OWL 2 Web Ontology Language Document Overview,* W3C, 2009.

[6] R. Cyganiak, D. Wood and M. Lanthaler, "RDF 1.1 concepts and abstract syntax," W3C, 2014.

[7] I. Horrocks, P. Patel-Schneider and H. Boley, "SWRL: A semantic web rule language combining OWL and RuleML," W3C Member submission, 2004.