Facilitating Policy Adherence in Federated OpenStack Clouds with Minimally Invasive Changes

Matthias Bastian, Max Plauth, and Andreas Polze

Operating Systems and Middleware Group
Hasso Plattner Institute for Digital Engineering
University of Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

Federated cloud setups can be constituted of a confusing amount of involved parties on both sides of providers and consumers. Hence, it is important to provide a policy mechanism that does not only allow providers to express their privacy policy statements but which also enables service consumers to express their own requirements how data should be treated by providers. However, integrating policy adherence for versatile policies necessitates profound alterations in existing cloud software platforms like OpenStack.

In this work, we demonstrate a prototypical approach that facilitates policy adherence support in OpenStack with minimally invasive changes. Employing certain concepts of the Aspect-Oriented Programming (AOP) paradigm, we present a framework that is able to inject policy adherence support in arbitrary sections of the OpenStack code base. With just a few lines of code, developers may add adherence support for policy attributes, even if the policy affects multiple OpenStack services.

1 Introduction

Cloud computing has become a pervasive method for running all kinds of applications. One of the major benefits is the possibility to acquire compute, storage and networking resources whenever they are needed. Recently, federated cloud scenarios have gained notable momentum, as they provide a viable alternative to public cloud providers [9]. Federated private cloud setups can often be comprised of many involved parties, both offering and consuming services. In such an unclear environment, it is crucial to provide means that allow providers to express their privacy policy statements, but which also enables service consumers to express their own requirements how data should be treated by providers. [5]

OpenStack is a well disseminated open source cloud platform that enjoys enormous popularity for driving many private cloud setups. One of the fundamental hurdles towards integrating proactive policy adherence into OpenStack is the tremendous implementation effort, as all services affected by a policy have to be altered significantly. We address this issue by presenting the *policyextension*-framework, which uses *PolicyExtensions* to facilitate policy adherence support with minimally invasive changes to the OpenStack code base. *PolicyExtensions* share many characteristics with plug-ins, as they are not part of the original code base.

In contrast to plug-ins, *PolicyExtensions* do not rely on plug-in mechanisms but inject their code at the locations of their own choice. The infrastructure for injecting *PolicyExtensions* is provided by the *policyextension*-framework.

Our approach provides the following characteristics:

- Interpretation of policies and the associated adherence mechanisms can be implemented in one single location, rather than being spread across the code base of numerous OpenStack services.
- Integrating policy support can be minimally invasive regarding code changes in existing services.
- Policy support code is easy to maintain.
- Facilities can be easily extended in order to support additional policy attributes.

The remainder of the paper is structured as follows: Section 2 provides an overview of prior approaches for providing policy adherence support in OpenStack. Furthermore, the employed policy language *CPPL* [6] is introduced. Afterwards, Section 3 elaborates on the fundamental design decisions that lead to the approach presented in Section 4. In Section 5, the applicability of the approach is demonstrated. Finally, Section 5 concludes this work.

2 Related Work

In this section, we provide an overview of prior policy integration concepts in OpenStack. We further introduce the policy language which is employed by the policy adherence mechanisms presented in this paper.

2.1 Policy Support in OpenStack

Here, we provide a brief overview of approaches for supporting policies in OpenStack that existed prior to this work.

2.1.1 oslo.policy

The *Oslo* project offers a plentitude of infrastructure utilities, e.g. for facilitating database access or for providing message queues. Also part of the project, *oslo.policy* is a dedicated library for managing inter-project communication. The library defines a format for specifying rules and policies and provides a corresponding policy execution engine. However, this policy engine does not suffice the requirements of federated private cloud setups, as *oslo.policy* is mainly intended to be used for authorization purposes. Potentially, *oslo.policy* can be used to guard other request properties for which *permit-or-deny* semantics are sufficient. However, more complex policies ought to be supported.

2.1.2 Swift Storage Policies

Swift is the OpenStack project for providing object storage. Policy support is provided at the fundamental level, as the containers holding objects can be annotated with policies such as replication rate. Policies are defined by users during the creation of a container, however policies are restricted to core concerns of the object storage and may not be used by other OpenStack projects.

2.1.3 Policy-based Scheduling in Nova

Upon creation of a new virtual machine, OpenStack has to decide on which host the VM should be instantiated. The *Nova* scheduler therefore attempts to locate a host that fulfills several predefined policies. Using this policy-based scheduling approach enables restricting VM instantiation requests to certain hosts [7]. Custom policies can be added by OpenStack instance operators by extending the policy set employed by the *Nova* scheduler. The major disadvantage of this approach is that users can neither review nor edit the policies that are applied to requests. Only OpenStack operators are able to add, review or edit policies. Another shortcoming of this approach is that the policy support is restricted to the *Nova* project.

2.1.4 Congress

The *Congress* project is a dedicated OpenStack project that aims at providing a centralized policy component for enabling compliance in cloud-based environments. Congress employs a monitoring approach in order to maintain a high degree of independence among OpenStack projects. It detects policy violations in a passive mode of operation by querying the state of all involved OpenStack services in regular intervals using their corresponding APIs. In case the state of an OpenStack service deviates from a policy, the violation is logged and notifications can be triggered.

One major limitation of *Congress* is that its monitoring-based approach impedes the implementation of proactive adherence mechanisms. Policies may specify how violations should be treated. In addition to logging violations and triggering notifications, policies can also be configured to revert policy violations. However, several actions are hard to revert, especially in cases where the violating action triggers many side-effects that have to be reverted as well. Potentially, proactive policy adherence based on *Congress* can be implemented. However, this would necessitate significant changes to the code base of all involved OpenStack services.

2.2 Compact Privacy Policy Language (CPPL)

In the context of the *Scalable and Secure Infrastructures for Cloud Operations* (SSICLOPS) project[1], many policy languages, including XACML [4], C^2L [13] and S4P [2], have been examined with regard to their applicability in federated cloud computing scenarios. [3] This survey revealed that none of the evaluated approaches

[1] `https://ssiclops.eu`

sufficiently satisfied the extensive list of assessment criteria. In consequence of these findings, the *Compact Privacy Policy Language* (CPPL) [6] has been designed to address expressiveness and extensibility, but at the same time *CPPL* addresses the mostly neglected requirements of runtime performance as well as communication and storage efficiency. In the scope of this work, one key attribute of *CPPL* is the very compact binary representation of its policy annotations, which facilitates efficient processing and enables the use of policy annotations at a per data-item level.

3 Design Considerations

In this section, we provide a brief discussion of the most crucial aspects that strongly influenced the design of the architecture presented in Section 4 for integrating policy support in OpenStack.

3.0.1 Monitoring versus Proactive Adherence

As elaborated in the context of *Congress* (see Section 2.1.4), proactive adherence to policies requires numerous changes in the OpenStack code base compared to a monitoring-based approach. However, the proactive approach never allows policy violations to occur in the first place, whereas monitoring-based approaches are limited to reacting to policy violations by means of logging and issuing counteracting actions. Here, we aim for the proactive approach.

3.0.2 Versatility of Policies

The *SSICLOPS* policy language *CPPL* (see Section 2.2) supports a wide range of policy attributes. Due to this versatility of *CPPL*, policies can affect any OpenStack project. In order to deal with this high degree of versatility, each OpenStack service might have to be adapted in order to support *CPPL*. As a result, an implementation strategy is required that allows for the integration of policy support with minimally invasive changes to the OpenStack code base.

3.0.3 Development Process of OpenStack

OpenStack projects are strictly separated in order to prevent inter-service dependencies. This level of isolation is also reflected by the development process: Projects may only interact using their regular, public APIs. Contributing code to OpenStack projects involves a very complex workflow [10] [12], which goes far beyond the usual habits of the typical fork-merge workflow applied on many GitHub projects. As this process introduces a fair amount of complexity, it is not feasible to perform changes across many OpenStack projects, as it would be necessary in order to implement adherence even to simple policies. Therefore, a central requirement for the presented approach is to keep the overhead for implementing policies as low as possible.

4 Approach

Our approach is comprised of two major components, the *policymiddleware* and the *policyextension*-framework. Both are documented hereinafter.

4.1 *policymiddleware*-Component

One of the central concepts of *CPPL* is, that policy annotation occurs at a per data-item level. Hence, any OpenStack component needs to be able to access policy information in order to further process it. Extending the OpenStack APIs with policy support is not an option, as it contradicts the design goal of implementing policy support with minimally invasive changes to the OpenStack code base. The effort required for adapting higher level APIs is manageable, however since higher level requests (e.g. to the *Horizon* dashboard) trigger many subsequent requests to lower level APIs, the consequent necessity for code changes affects the entire OpenStack code base.

In order to make policy information transparently available to arbitrary OpenStack components, we introduce a new middleware component called *policymiddleware*, which is based on the general concepts of the *keystonemiddleware* component. To unburden other OpenStack components of the task of user authentication, all requests pass through the *keystonemiddleware*, which verifies the validity of the `X-Auth-Token` in the HTTP headers first. Requests with invalid authentication tokens are rejected right away and never reach the intended service, whereas valid requests are annotated with user credentials such as username and the user identifier. In a similar fashion, the *policymiddleware* validates incoming *CPPL* annotations and deposits the policy information in *Keystone*, from where it can be retrieved from the *policyextension*-framework. In cases where incoming requests are not annotated with policies in the first place, the *policymiddleware* fetches user-based policy information from *Keystone* and annotates the request with the resolved policies. The mechanics of both the *keystonemiddleware* and the *policymiddleware* are illustrated in Figure 1.

4.2 *policyextension*-Framework

With the *policymiddleware* providing policy information, the logic for interpreting and adhering to policies is still required. To close this gap, we introduce the *policyextension*-framework, which enables developers to implement policy support through so-called *PolicyExtensions*, which share certain properties with plug-ins. Unlike plug-ins, however, they do not rely on potentially missing extension facilities but rather inject their logic through *monkey patching* mechanisms, where the framework modifies the behavior of selected classes and functions at runtime. Even though monkey patching is often considered to be an obscure method, we employed it anyway to compensate for the lack of a unified plug-in infrastructure across OpenStack projects. Furthermore, with proper safeguards like version checking in place,

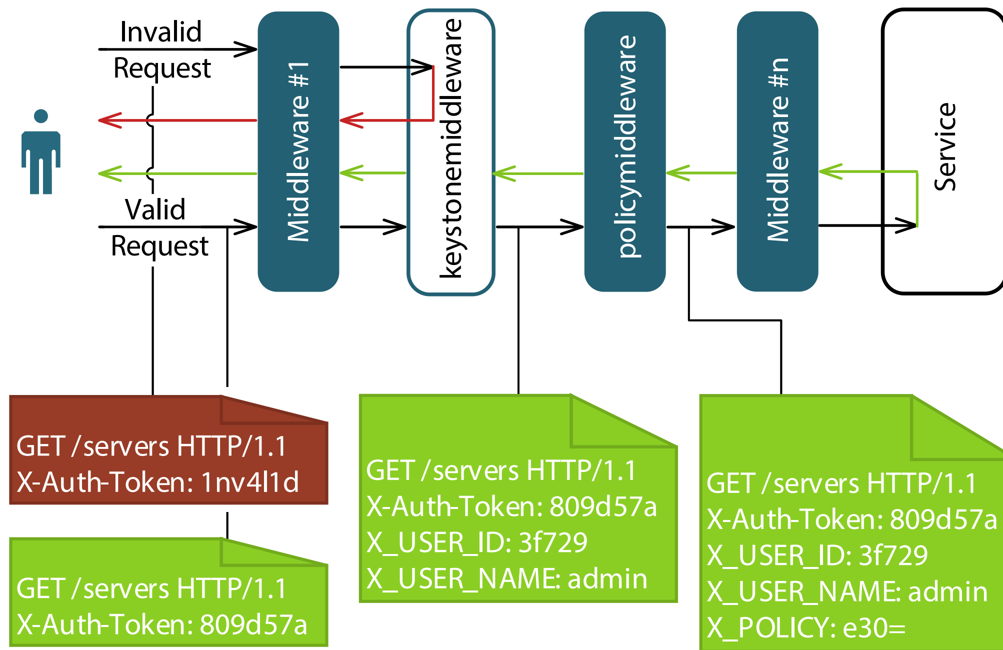

Figure 1: Analogous to *keystonemiddleware*, the *policymiddleware* component transparently annotates requests with policy information.

monkey patching is even used in production code. The *Nova* project for example employs *monkey patching* indirectly by employing the *eventlet* library [11].

Developers can implement new *PolicyExtensions* by creating a new class that inherits from the base class `PolicyExtensionBase`. In this new class, the attribute `func_paths` has to be provided, which should contain a list of functions to be modified. Furthermore, a method with the same name has to be implemented, containing the logic that is executed prior to the execution of the original function. The *policyextension*-framework handles the entire patching process, which is visualized in Figure 2. It furthermore provides access to the arguments that the original function is called with. As the policy information is initially only available at the API endpoints of each service, the framework also makes this information available from arbitrary locations in the service implementations. Last but not least, the framework converts incoming policy information to *Python*-friendly dictionaries, which can be easily queried. Since this format conversion mechanism employs a well-defined interface for decoding and encoding policy information, support for other policy annotation languages can be easily retrofitted.

The mechanisms provided by the *policyextension*-framework implement many concepts of the *Aspect Oriented Programming* (AOP) paradigm [8]. Mapping the components of the presented framework to AOP concepts, `PolicyExtension` classes resemble *Aspects* whereas the original functions to be modified correspond to *Jointpoints* and the `func_paths` represents the *Pointcut*. The method of a `PolicyExtension` class corresponds to an *Advice*.

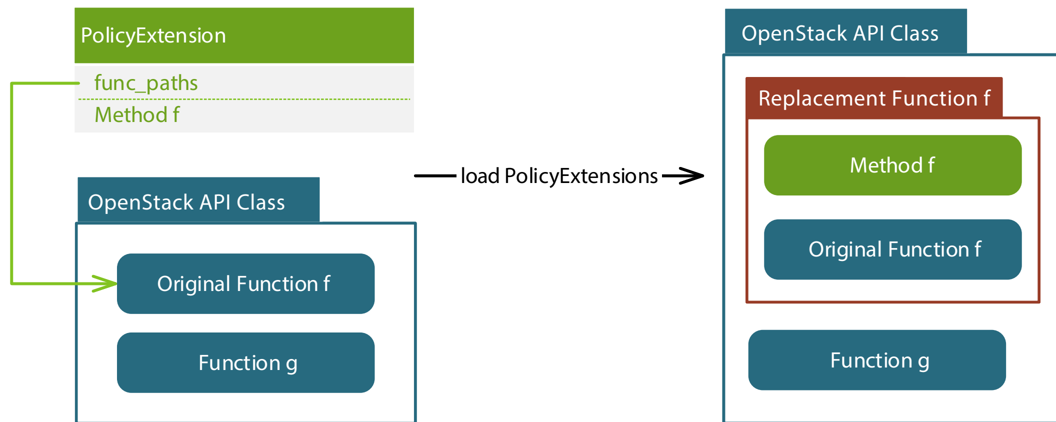

Figure 2: *PolicyExtensions* dynamically modify the behavior of OpenStack classes and functions at runtime by applying *monkey patching*.

5 Applicability

Here, we demonstrate the applicability of the approach presented in Section 4 by providing example code for supporting two policy attributes.

5.1 Enforcing Disk Encryption

The *policyextension*-framework uses the dictionary data type `dict` in *Python* to provide the policy information. Listing 1 shows the `dict`-based representation of a policy that requires the use of disk encryption.

Listing 1: `dict`-based policy notation for enforcing disk encryption:

```
1 {
2   "storage": {
3     "encryption": True
4   }
5 }
```

To support adherence to such a policy, the *Cinder* service of OpenStack has to be adapted. In OpenStack, the Cinder service is responsible for providing *Block Storage*, which is indicated by the `storage` key. The embedded key `encryption` with the corresponding boolean value `true` then specifies, that newly created volumes must use encryption. Below, the example code demonstrates how the proactive policy adherence can be implemented by creating a new *PolicyExtension*:

Listing 2: Example *PolicyExtension* for enforcing disk encryption policies:

```
1 from policyextension import PolicyExtensionBase, PolicyViolation
2 from cinder.volume import volume_types
3
4 class CinderEncryptedVolumeTypeRequiredExtension(PolicyExtensionBase):
```

```
5   func_paths = ['cinder.volume.api.API. create ']
6
7   def create(self, func_args, policy):
8     try:
9       if policy['storage']['encryption']:
10        volume_type = func_args['volume_type'] or volume_types.
              get_default_volume_type()
11        if not volume_type or not volume_types.is_encrypted(func_args['
              context'], volume_type['id']):
12          msg = "Your policy requires using an encrypted volume type ."
13          raise PolicyViolation(msg)
14    except KeyError:
15      pass
```

5.2 Restriction of Availability Zones

As a second example, we demonstrate the code for supporting adherence to a policy that restricts the instantiation of virtual machines to a set of whitelisted availability zones. Here, we are using the OpenStack mechanism of availability zones in order to model geographic locations. Under the assumption that an *Availability Zone* with the name az2 exists, the human readable representation of the policy for this example is demonstrated in Listing 3.

Listing 3: `dict`-based policy notation for restricting availability zones:

```
1 {
2   "availability_zones": ["az2"]
3 }
```

Since we are using availability zones to model geographic locations of the data center, we decided to employ a whitelist approach in order to ensure that services are only instantiated in a region that is explicitly approved by the user. The resulting implementation is presented in Listing 4.

Listing 4: Example *PolicyExtension* that can restrict the execution of virtual machine instances to whitelisted availability zones:

```
1  from policyextension import PolicyExtensionBase, PolicyViolation
2  import random
3
4  class AvailabilityZoneRestrictionExtension(PolicyExtensionBase):
5    func_paths = ['nova.compute.api.API. create ']
6
7    def create(self, func_args, policy):
8      availability_zone = func_args['availability_zone']
9      try:
10       az_whitelist = policy['availability_zones']
11       if availability_zone:
12         if availability_zone not in az_whitelist:
13           msg = ("Your policy does not allow the availability zone you selected .")
```

```
14          raise PolicyViolation(msg)
15        elif az_whitelist:
16          func_args['availability_zone'] = random.choice(az_whitelist)
17      except KeyError:
18        pass
```

6 Conclusion

In this paper, we presented the *policyextension*-framework, which uses *PolicyExtensions* to facilitate policy adherence support in OpenStack with minimally invasive changes to the code base. To demonstrate the applicability of the framework, we discussed exemplary implementations that provide policy adherence support for enforcing disk encryption and for restricting virtual machines to whitelisted availability zones. Both implementations employ *permit-or-deny* semantics. As next steps, we would like to evaluate modifying semantics, where incoming requests are modified in order to satisfy policy requirements. Furthermore, we are planning to evaluate our approach in a federated OpenStack testbed.

All results are based on the master's thesis by Matthias Bastian. Additional implementation details have been documented in the original thesis [1].

Acknowledgement & Disclaimer

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] M. Bastian. "Design and Integration of a Framework for Enforcing User-Defined Policies in OpenStack". Master's Thesis (in German). Hasso Plattner Institute for Software Systems Engineering, University of Potsdam, Jan. 2017, page 88.

[2] M. Y. Becker, A. Malkis, and L. Bussard. "A Practical Generic Privacy Language". English. In: *Information Systems Security*. Edited by S. Jha and A. Mathuria. Volume 6503. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 125–139. ISBN: 978-3-642-17713-2. DOI: 10.1007/978-3-642-17714-9_10.

[3] F. Eberhardt, M. Plauth, A. Polze, S. Klauck, M. Uflacker, J. Hiller, O. Hohlfeld, and K. Wehrle. *SSICLOPS Deliverable 2.1: Report on Body of Knowledge in Secure Cloud Data Storage*. Technical report. June 2015.

[4] S. Godik, A. Anderson, B. Parducci, P. Humenn, and S. Vajjhala. *OASIS eXtensible Access Control Markup language (XACML)*. Technical report. OASIS, May 2002.

[5] M. Henze, M. Großfengels, M. Koprowski, and K. Wehrle. “Towards Data Handling Requirements-Aware Cloud Computing”. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Volume 2. Dec. 2013, pages 266–269. DOI: 10.1109/CloudCom.2013.145.

[6] M. Henze, J. Hiller, S. Schmerling, J. H. Ziegeldorf, and K. Wehrle. “CPPL: Compact Privacy Policy Language”. In: *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*. WPES ’16. Vienna, Austria: ACM, 2016, pages 99–110. ISBN: 978-1-4503-4569-9. DOI: 10.1145/2994620.2994627.

[7] Khanh-Toan Tran and Jérôme Gallard. *A new mechanism for nova-scheduler: Policy-based Scheduling*. Technical report. 2013.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-oriented programming”. In: *ECOOP’97 – Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*. Edited by M. Akşit and S. Matsuoka. Berlin, Heidelberg: Springer, 1997, pages 220–242. ISBN: 978-3-540-69127-3. DOI: 10.1007/BFb0053381.

[9] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. “Iaas Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures”. In: *Computer* 45.12 (2012), pages 65–72.

[10] OpenStack Foundation. *Developer’s Guide*.

[11] OpenStack Foundation. *Nova Threading Model*.

[12] OpenStack Foundation. *OpenStack Blueprints*.

[13] J. Poroor and B. Jayaraman. “C2L:A Formal Policy Language for Secure Cloud Configurations”. In: *Procedia Computer Science* 10 (2012). {ANT} 2012 and MobiWIS 2012, pages 499–506. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2012.06.064.