Cloud native Authorization

delegating authorization decisions from microservices to Open Policy Engine

PRASHANNA RAI

Master’s programme in Computer Science
Date: August 3, 2023

Supervisor: Vanja Divkovic
Examiner:

Host company: Ericsson

Abstract

Despite companies got speed in development and deployment process with adoption of Cloud native development approach, teams were giving significant time and energy in solving recurring problem of authorization logic associated in every microservices. Those solution were very specialized to certain applications or structure of ACL representations which are very hard to reuse with other applications. This has increased development and maintenance costs with no standardization of authorization logic.

With the use of Open policy Agent as General purpose policy engine over our existing old authorization logic, we were able to express the coarsed-grained authorization and fine-grained authorization logic using rego language that produces indentical authorization decisions like old systems had.

Regardless of what techincal stack our service relied on, versatility of OPA allowed us to unify all the authorization logic with one programming language knowns as Rego. It was observed that there was almost small difference in latency i.e 1.109 times slower in latency when service was integrated with OPA as SideCar [1]. But in the case of fine-grained authorization with large data, Wasm based system integratation performs 1.30 times faster than old authorization system.

Among all the approaches way of integratation of OPA with service, we found that Distributed PDP i.e integratation of OPA via Unix domain socket had lower overhead and faster response than Central PDP i.e integratation of OPA via HTTP with service.

Keywords

Cloud native authorization, Open Policy Agent, General-purpose policy engine, Centralized Authorization, Distributed Authorization, Role-based access control, Security

Acknowledgments

I would like to acknowledge my supervisor Vanja Divkovic and my reviewer Konstantinos Sagonas for all help and feedback they have given me in this

thesis project. I would also like to thank the Ericsson team for providing support when needed.

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

ACL Access control list

ADP Application Develop ment Platform

CNIS Cloud Native Infrastructure

DSL Domain Specific Language

EMC Equipment Manager Centralized

ICT Information and Communication Technology

NFVI Network Functions Virtualization Infrastructure

OMC Operations Manager Cloud infrastructure

OPA Open Policy Agent

PAP Policy Administration Point

PDP Policy Decision Point

PEP Policy Enforcement Point

PV Persistent Volume

RAN Radio Access Network

RBAC Role-based access control

REST Representational state transfer

TBAC Target-based access control

XACML eXtensible Access Control Markup Language

Chapter 1

Introduction

According to Cloud native Computing foundations definitions [2], Cloud native is an approach to develop applications using a stack of technologies which are autoscale, vendor-agnostics applications and Self-service deployed. Cloud native application can be identified by Microservices, container-based applications, domain Functionality Segregation and dynamic orchestration of compute, storage, and networking resources. In short, Cloud is about where software runs. Cloud native is about how it runs.

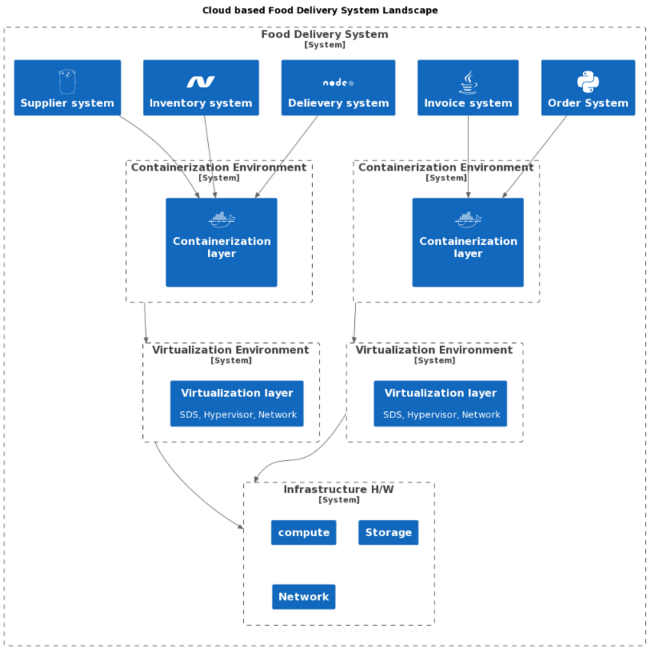


Figure 1.1 – Cloud based microservices

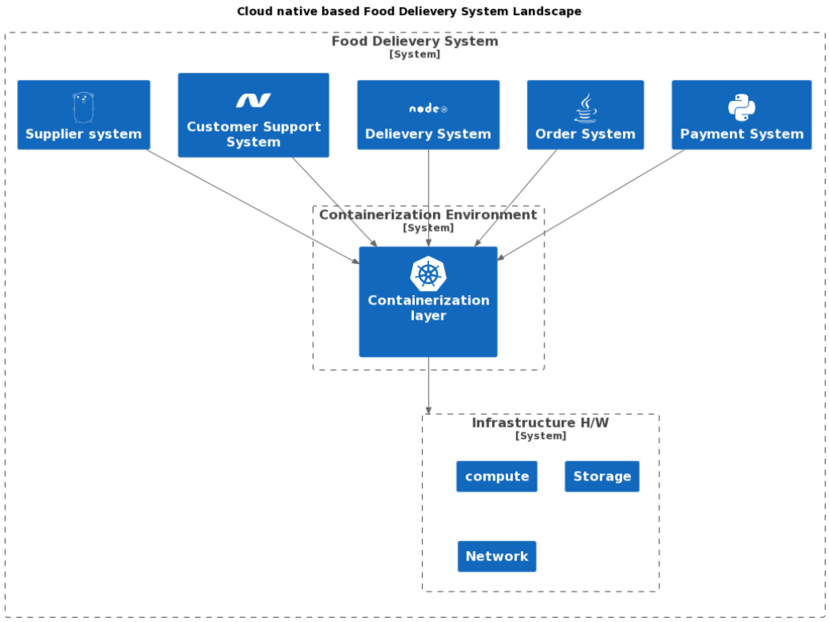


Figure 1.2 – Cloud native based microservices

With the demand for faster, more frequent delivery and on-demand scaling out, a lot of companies have shifted to cloud native software development. Telecommunication network providers like Ericsson are embracing cloud native solutions where everything that can benefit from the cloud, will run in the cloud. Ericsson is one of the leading providers of Information and Communication Technology (ICT) to service providers and many of our latest solutions including 5G Core and Radio Access Network (RAN) are cloud based. Ericsson has a long history of systematically incorporating security and privacy considerations into all relevant aspects and phases of our product value flow. Every product, therefore, needs to protect the privacy of users and its resources against unauthorized access and must still provide full functionality. Individual parts of the entire application no longer need to run on the same computer, which makes it easier to scale the entire system, but also means that the applications need to be configured to communicate with each other. Seamlessly secure networks rarely exist, and security must be considered from the initial steps of product development until the end of the product's life cycle and even further in the life cycle management processes. Ensuring the security of software in the highly dynamic and ever-changing security landscape is a continuous and evolving process.

Ericsson provides private cloud solutions which comes with the challenges of managing users or services and protecting cloud resources like services. Ericsson started moving to cloud native approach, those applications are deployed into private cloud where Kubernetes cluster are hosted. The existing application design embeds authorization logic into each service which means that for each applications, different development teams might be solving the same problem in different ways. The representation of policy and making decision based on existing policy is not commong across different development teams. With the growing number of services, testing and maintaining an authorization logic becomes tedious and error-prone especially when we want to add new roles or introduce new policies into applications. If we want to update some roles of applications then we need to make changes to the source code, rebuild all services, release them, and redeploy. This creates a hassle with development time for managing a large microservices ecosystem. An access control policy are embeded into source code which has made testing more costly since the tester needs to have an idea of source code of protected applications works. In a typical Ericsson cloud-native application, application code is coupled with Policy Decision Point (PDP) and Policy Enforcement Point (PEP) and every service implements its solution. Policy Administration Point (PAP) is usually not supported, and policies are built into the service images. The consequence is that users don't have a view into authorization policies and can't adapt them to their own needs. It is also difficult for application developers and testers to understand authorization policies without digging into the code of different services. Since authorization policies are coupled with the application code they cannot be tested separately. There is no way to administer policies since they are hard coded and hidden from the users.

1.1 Purpose

The thesis project investigated expressiveness of rego language to solve coarsed-grained authorization and fine-grained authorization logic. The scope of project was extended to measure latency involved with integratation approaches of Open Policy Agent (OPA) with Java based application and Open Policy Agent (OPA) with Python based application. Depending upon integration approach of OPA, we documented security risks associated with OPA by using thread modeling approach.

1.2 Goals

The thesis validated with expressiveness of rego language to fufill coarsed-grained and fine-grained authorization based on Role-based access control (RBAC). Then we investigate performance of system with different integration approach of Open Policy Agent (OPA) with service. We study behaviour of performance when using optimization flag and rule-indexing in OPA compiler. In first approach, the experiment was performed by deploying OPA as REST API server (centralized Policy Decision Point (PDP)) into different node which is not used by calling service, decisions are consumed by EMC application via HTTP. In second approach, OPA is run as a sidecar (distributed Policy Decision Point (PDP)) or Wasm binary is embeded within service or use Intermediate file with Jarl library.

1.3 Delimitations

The work was focused on a Cloud native application named EMC and OMC running on Kubernetes. we didn't analyze communication with external applications or components. The underlying infrastructure and execution environment were also out of scope.

Chapter 2

Background

2.1 Coarsed-grained authorization

Authorization is process of making decisions to authenticated users should be allowed perform operations. Depending upon the level of protection, Authorization can be roughly divided into two categories: coarse-grained authorization and fine-grained authorization. Coarsed-grained authorization is ability to protect resource before authenticated users is allowed to perform operations. Coarsed-grained authorization is achieved by the implementation of a Pre-filter. Pre-filter ensures authorization rules are checked before the method call. For example: If the request to url ”A” is protected then it can be defined as coarse-grained”authorization.

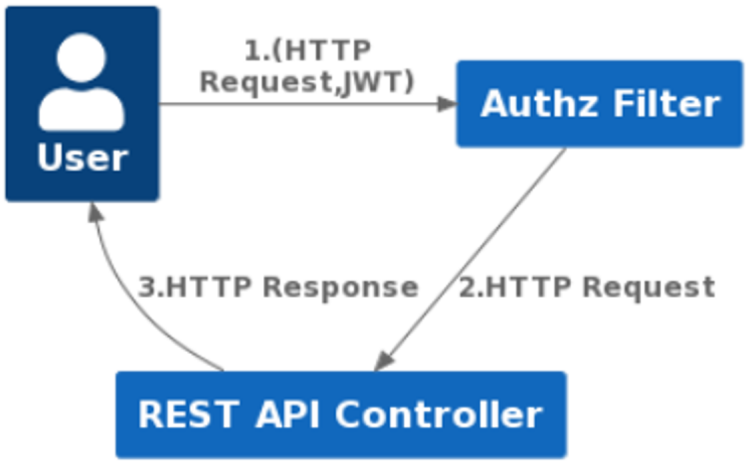


Figure 2.1 – Coarsed grained authorization process

2.2 Fine-grained authorization

While fine-grained authorization typically refers to the selective availability of the portions of a page, rather than restricting access to a page entirely[3]. For example: In case of fine-grained authorization, website hides admin page from authenticated user if user doesn't belong to admin. However, fine-grained authorization can be identified by the involvement of the post-filter. Post-Filter ensures authorization rules after the method executes [4]. The post-filter aspect filters from the returned collection or array those elements that don't follow your rules. If the rules aren't respected, instead of returning the result to the caller, it returns empty. For example: In Online Shopping, if a vendor wants to list products that he owns, then vendors should only get products that are owned by the vendor. This is done by post-filter [5].

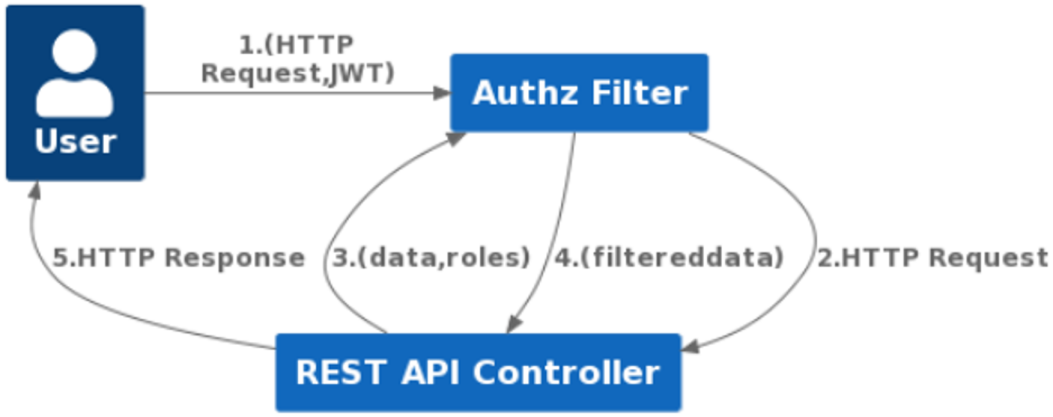


Figure 2.2 – Fine grained authorization process

2.3 Policy

Policy is document containing rules basis on the attributes of the subject, object, and possible environment conditions.

2.4 Access-Control List

An access-control list is a way to represent permissions associated with a resources. Ericsson uses two ways to represent Access control list (ACL) list

namely Role-based access control (RBAC) and Target-based access control (TBAC).

2.4.1 Role-based access control

In RBAC[6], Permissions are assigned to roles, and then roles are assigned to users. In this approach, system makes decision basis on roles of an authenticated user.

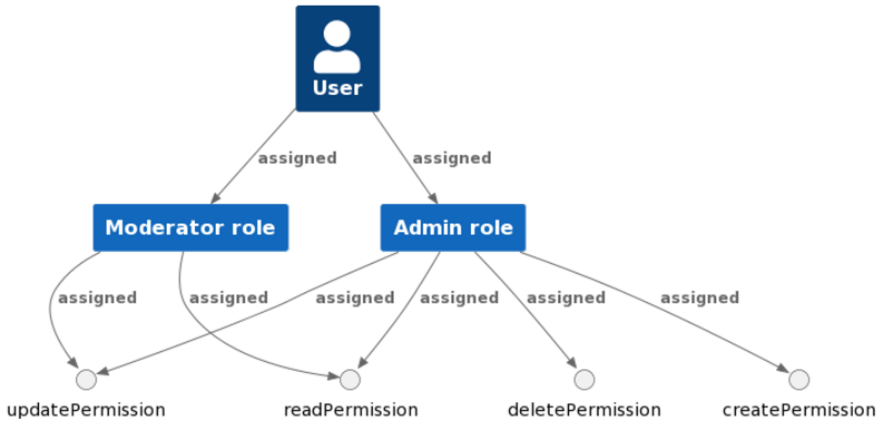


Figure 2.3 – RBAC

The user having "Read" role can only execute read operation on "v1/Ressources". An example of RBAC based ACL is as follows.

```
{
  "Roles": [
    {
    "RoleName": "Create",
    "Permissions": [
      "createResource",
      "updateResource"
    ]
    },
    {
    "RoleName": "Delete",
    "Permissions": [
      "DeleteResource"
    ]
    },
```

```
    {
    "RoleName": "Read",
    "Permissions": [
     "ListResource",
     "GetResource"
    ]
    }
  ]
  }
```

```
{
  "Statements": [
    {
    "Method": "GET",
    "Resource": "v1/Resources",
    "Permission": "ListResource"
    },
    {
    "Method": "GET",
    "Resource": "v1/Resources/{id}",
    "Permission": "GetResource"
    },
    {
    "Method": "POST",
    "Resource": "v1/Resource/",
    "Permission": "createResource"
    },
    {
    "Method": "PUT",
    "Resource": "v1/Resource/{id}",
    "Permission": "updateResource"
    }
  ]
  }
```

2.4.2 Attribute-based access control

In ABAC [7] , access control decisions are made dynamically basis on following attributes.

- Attributes about the subject; that is, the user making the request. This could include their username, any groups they belong to, how they were authenticated, when they last authenticated, and so on.
- Attributes about the resource or object being accessed, such as the URI of the resource or a security label (TOP SECRET, for example).
- Attributes about the action the user is trying to perform, such as the HTTP method.
- Attributes about the environment or context in which the operation is taking place. This might include the local time of day or the location of the user performing the action.

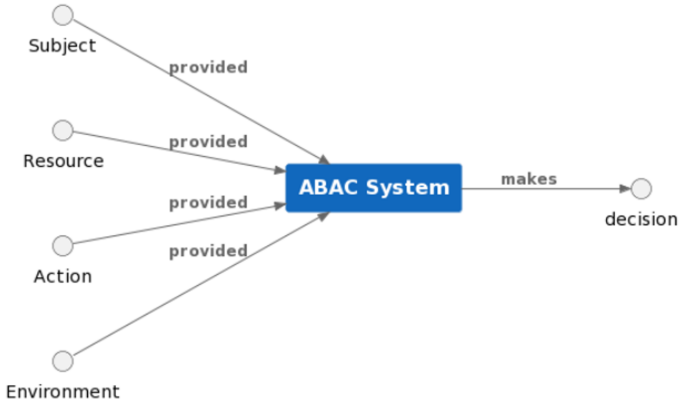


Figure 2.4 – ABAC

2.4.3 Target-based access control

Target-based access control is a subset of Attribute-based access control where It is based on the target concept. This controls which nodes a user has access to and which roles the user has when accessing a node. Each node stores a list of targets. Target is the logical identifier of a node or a group of nodes in the network. It distinguishes the particular node from others in the network and provides the possibility to gather nodes according to their logical function and/or other criteria such as geographical location.

2.5 Architecture of access-control systems

As described in the XACML model, the core components consist of the following components:

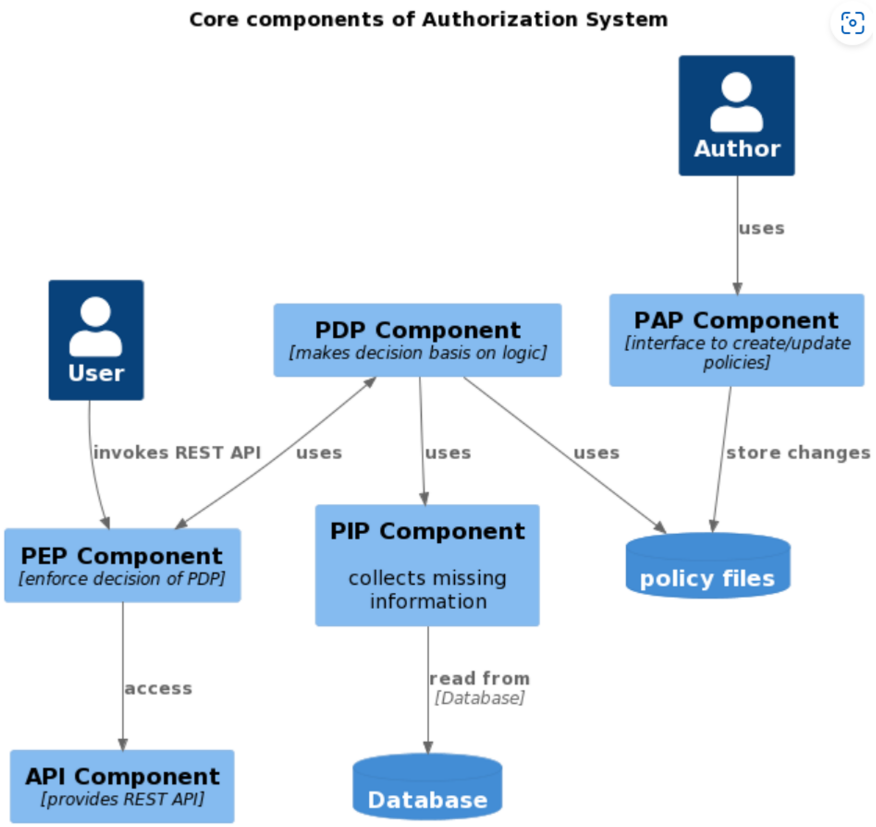


Figure 2.5 – Core components of Authorization System

2.5.1 Authorization policies

It define conditions under which a user can be granted access to a certain resource. In our experiment, we would express all the policies in Rego.

2.5.2 Policy store

It is storage for authorization policies. It can be stored in git, filesystem, PERSISTENT VOLUME (PV) or a service that's exposed via REPRESENTATIONAL STATE TRANSFER (REST) .

2.5.3 Policy Decision Point(PDP)

It is the brain of the authorization system. It loads policies from the policy store and makes authorization decisions at the time of the request. In our experiment, all the responsibilities are fulfilled by OPA [8].

2.5.4 Policy Enforcement Point

PEP intercepts all requests and enforces authorization policies. It makes contact with the PDP to obtain the access decision.

2.5.5 Policy Information Point

PIP helps to resolve information required by the policy to make decision. These pieces of information are called Attributes. The PIP is typically composed of many Attribute Value Providers, each providing a variety of information. For example, there may be a value provider that provides user identity claim attributes; another may provide information about the purchase order limit of the requester. The attributes delivered may come from a relational database, requests to REPRESENTATIONAL STATE TRANSFER (REST) APIs, LDAP, or any other source of information.

2.5.6 Policy Administration Point

Policy Administration Point writes the policies to a policy store.

JSON Web Tokens

All the HTTP operations are stateless that means we need to repeatedly do authentication for every access to protected API. This might be resource consuming and time-consuming process. To overcome this problem, timelimit tokens are exchanged after authentication that contain some information about the client. But these tokens need to store in database. database cannot scale in performance with a huge level of traffic. Rather than storing the token state in the database, we can instead encode that state directly into the token ID and send it to the client. For example, we could serialize the token fields into a JSON object, which you then Base64url-encode to create a string that you can use as the token ID. When the token is presented back to the API, you then simply decode the token

and parse the JSON to recover the attributes of the session. JSON Web Tokens are standard formats for self-contained security tokens. A JWT consists of a set of claims about a user represented as a JSON object, together with a header describing the format of the token. JWTs are cryptographically protected against tampering and can also be encrypted. Example of JWT token is as follows

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIxQ3
ZFRNUk5oMEE0THNzMUpU....
```

When we decode token, we can get some useful information that can be used during authorization.

```
      Token header
      ------------
      {
        "typ": "JWT",
        "alg": "RS256",
        "kid": "1CwrdTMRNh0A4Lss1J..."
      }

      Token claims
      ------------
      {
        "acr": "1",
        "allowed-origins": [
          "https://epraria"
        ],
        "aud": "account",
        "azp": "OMC_GUI",
        "email_verified": false,
        "exp": 1681664803,
        "iat": 1681664503,
        "iss": "https://eric-sec-access-mgmt-http:8443/
        auth/realms/omc",
        "jti": "ebe587dc-d5bc-4327-bd09-c637acd9f1e0",
        "preferred_username": "createjob",
        "realm_access": {
          "roles": [
            "default-roles-omc",
            "offline_access",
```

```
          "uma_authorization",
          "CreateJob"
        ]
      },
      "resource_access": {
        "account": {
          "roles": [
            "manage-account",
            "manage-account-links",
            "view-profile"
          ]
        }
      },
      "scope": "email profile",
      "session_state":
        ↪ "aa7382f7-fbbd-486a-9712-1114b4b5c2a6",
      "sid": "aa7382f7-fbbd-486a-9712-1114b4b5c2a6",
      "sub": "56b604eb-af26-4e0e-8949-1ad57b17c185",
      "typ": "Bearer"
    }
```

Related work area

Open Policy Agent (OPA) , eXtensible Access Control Markup Language (XACML) specification, Speedle are some of the existing open-source software that provides solution of universal authorization engine.

2.5.7 XACML

XACML is standard developed by OASIS that describes both a policy language and an access control decision. The request and response are encoded in XML. JSON may be used for authorization queries and responses. The policy language is used to describe general access control logic. The request/response language lets you form a query to ask whether or not a given action should be allowed, and interpret the result. The response contains one of four values: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (the request can't be answered by this service). The adoption

of XACML lessened now. There are few open source implementation of XACML 3.0 some of them are Sun's XACML and Balana. The policy language model[9] specified by xacml 3.0 is very complex as this can be seen in figure

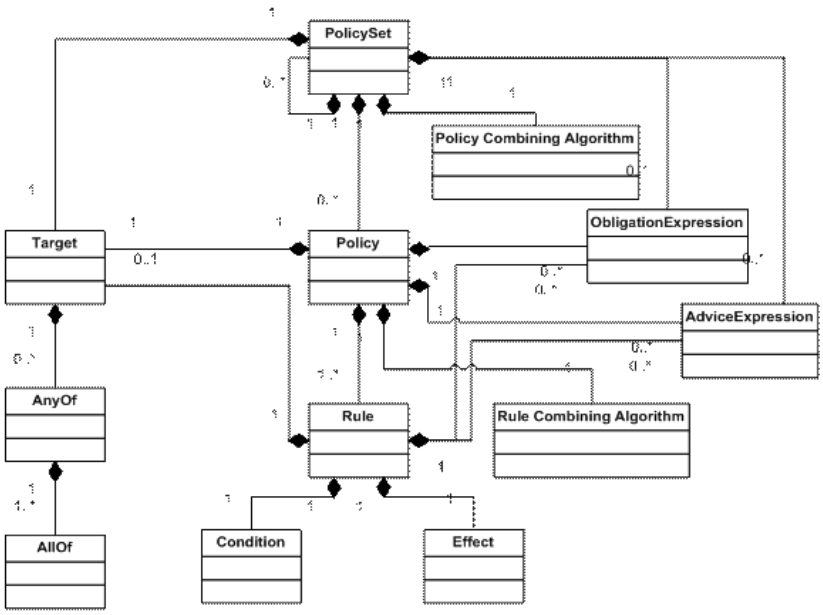


Figure 2.6 – policy language model of XACML

2.5.8 Speedle

Speedle was developed at Oracle, but didn't get much community adoption. So it is archived now. But team is working on Speedle+ project. It can be integrated only with Docker as Auth plugin, Kubernetes as authorization webhook, Istio as Istio Speedle webhook .

2.5.9 OPA

Open Policy Agent (OPA) is an open source, lightweight, general-purpose policy engine with no external dependency. OPA uses declarative language called Rego language. The declarative language allows policy author to focus on providing description of logic rather than how it should achieved. OPA has excellent support for loading JSON and YAML. OPA has a growing community and is easy to implement. Since, our access control list is defined in JSON file. It makes good candidate to consider OPA as solution for this problem.

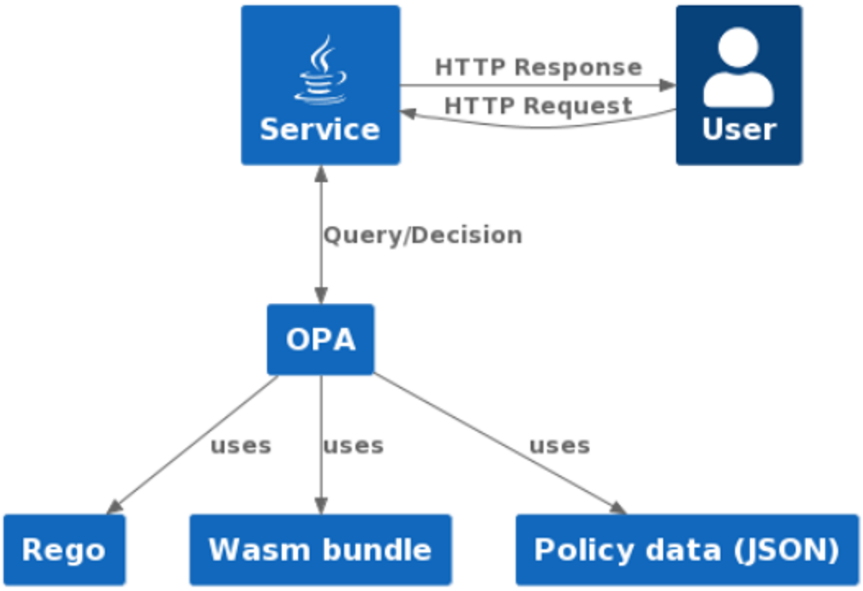


Figure 2.7 – OPA-Overview

OPA is known to be versatile authoization engine meaning that OPA has lot of integration capalilities like sidecar in istio, host-level daemon, REST API for any languages, or GO SDK with GO lang service, Wasm binary file with any languages.

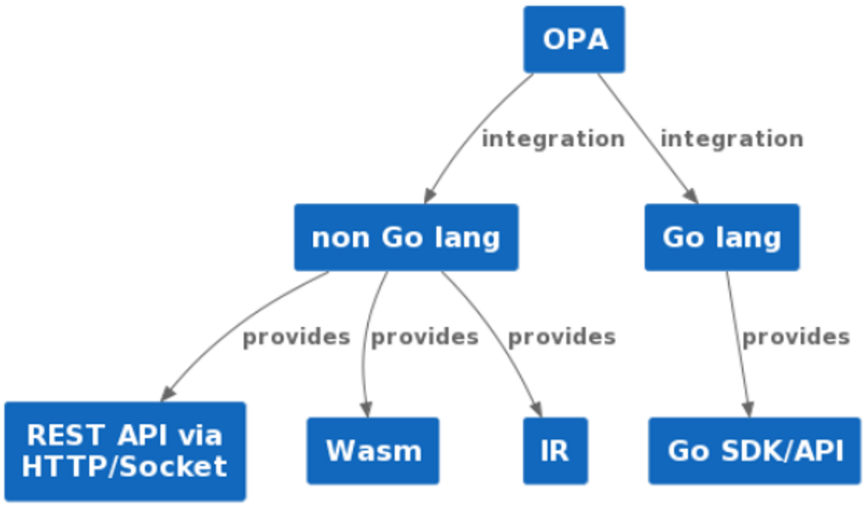


Figure 2.8 – integration avaiable with OPA

2.5.10 Rego

Rego[10] is a Domain Specific Language (DSL) created for OPA and can be used as the policy language in the OPA engine. Rego is declarative langauge so the policy authors can focus on what queries should return rather than how queries should be executed. It can be used to process JSON and YAML file without need of dependency. Example of rego is as follows:

```
1  package authz.v1.policy
2  import future.keywords
3  default allow = false
4  allow{
5     input.user.id == "ADMIN"
6  }
```

2.5.11 Optimization levels

For optimization, OPA compiler can compile rego file with 3 different optimization levels that aims to bring evalaution time from non-linear $\mathcal{O}(n * m)$ to almost constant time $\mathcal{O}(c)$. Here, c is time taken to lookup element in Trie data-structure, n and m refers to number of elements of arrays.

- optimization level 0 : with this flag, rego file remains unchanged.
- optimization level 1 : Rules containing array lookup are transform in such a way that each rule is created with each item of array so that rule can be evaluated without interating arrays. By doing so rules becomes easier to indexed thus by increases performance. Rules containing independent values are evaluated while rules containing unknown value (i.e value that are not known during compilation) are transfromed into virtual documents[11]. These depedent rules are not inlined. If a base or virtual document is targeted by a with statement in the policy, the document will not be inlined. For example: rules that needs array lookup during evaluation are transformed into new block of rules with equality expression. to be more specific, rules are that with roles
- optimization level 2 : It does all the transformation same as with -O=1 except virtual documents produced by rules that depend on unknowns may be inlined into call sites. In addition, more aggressive

inlining is applied within rules. This includes copy propagation and inlining of certain negated statements that would otherwise generate support rules[12].

2.5.12 Indexing

In recent version of OPA, OPA supports automatic rule indexing by building a data structure called Trie[13] from equality expressions contained in rule sets. During rule evaluation, OPA only traverses or evaluates subset of rules that satisfy input, resulting to faster evaluation of query. Due to use of rule-indexing, evaluatation time of query doesn't get affect with growth of size of policy.

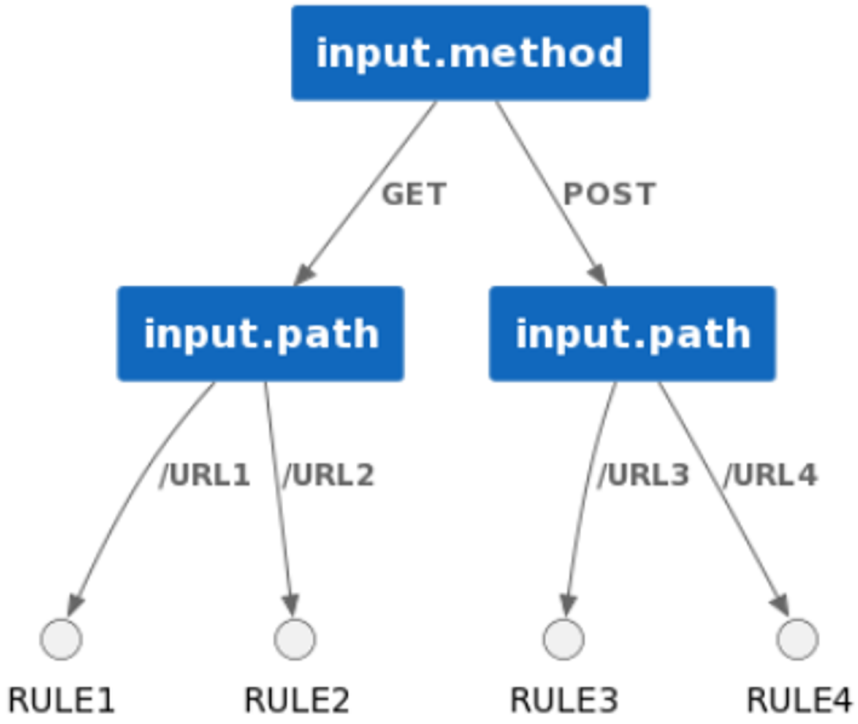


Figure 2.9 – Trie data structure

2.5.13 Partial evaluation

Rules containing known value can be indexed automatically. For example, Rules with following equality expressions can be indexed.

- "GET" == input.method
- input.url =="items"
- glob.match(<pattern>,<delimeter>,<indexed ref>)

Here, indexed ref can be defined as follows

- A base document e.g input
- <indexed ref>.<literal>
- <indexed ref>[<literal>]

But, Rules with following equality expressions cannot be indexed.

- regex.match(policy.url,input.url)
- upper("post") == input.method
- input.role ==input.jwt.role

In rego, it is possible to express rule with unknown values. The unknown values will be passed during evaluation as input. So those kinds of rules can take unknown time.

By partially evaluating policies, OPA can perform computation at compile time instead of runtime.

```
1   package api
2   default allow = false
3   allow = true {
4      rule = rules[_]
5      input.method = rule.method
6      input.resource = rule.resource
7      input.role = rule.role
8   }
9   rules = [
10     {"role": "moderator","method": "PUT", "resource":
          ↪ "account"},
11     {"role": "admin","method": "DELETE", "resource":
          ↪ "account"},
12     {"role": "user","method": "READ", "resource":
          ↪ "account"},
13  ]
```

To get answer of the policy query, OPA needs to the all items of rules to find a match each time an API request is received. As we keep on adding new api, rules also grows and evaluation takes longer time.

```
package api
default allow = false
allow = true {
   input.method = "PUT"
   input.resource = "account"
   input.role = "moderator"
}
allow = true {
   input.method = "DELETE"
   input.resource = "account"
   input.role = "admin"
}
allow = true {
   input.method = "READ"
   input.resource = "account"
   input.role = "user"
}
```

if we are able to express rule in unrolled version, then opa can index rules. OPA can automatically transform to unrolled version for rules containing independent of input (all the knowns values). This is features is called partial evaluation. With partial evaluation features, partially evaluated policy at runtime are cache for later.

Currently, partial evaluation is supported either by OPA (GO based) SDK or OPA REST api server. To enable partial evaluation in embed library To enable partial evaluation feature in OPA during evaluation of query, service needs to send query parameter called partial.

```
curl localhost:8181/v1/data/api/allow?partial -d
   ↪ @req.json
```

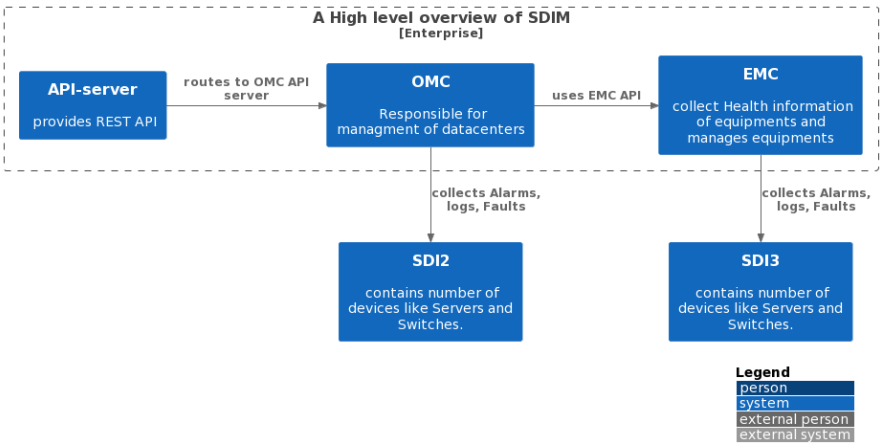
2.6 Existing Service

The project involves with working with existing services named Equipment Manager Centralized (EMC) and Operations Manager Cloud infrastructure (OMC) service. EMC application is developed with Java language and

is deployed to Kubernetes cluster. While OMC application is developed with Python language. EMC is part of OMC application. In production environment, OMC application will execute api of EMC service.

2.6.1 OMC

Operations Manager Cloud infrastructure (OMC) [14] is a cloud native application that simplifies operating Ericsson Cloud infrastructure solutions (Network Functions Virtualization Infrastructure (NFVI) and Cloud Native Infrastructure (CNIS)) by providing capabilities as performing lifecycle management operations from a centralized location and collecting faults, metrics, logs and events from remote sites. OMC is developed from the Ericsson Application Develop ment Platform (ADP). ADP is a collection of reusable common components like building block for developing Ericsson cloud native applications.



(a)

Figure 2.10 – Context Diagram of SDIM
This diagram uses C4 model notations.

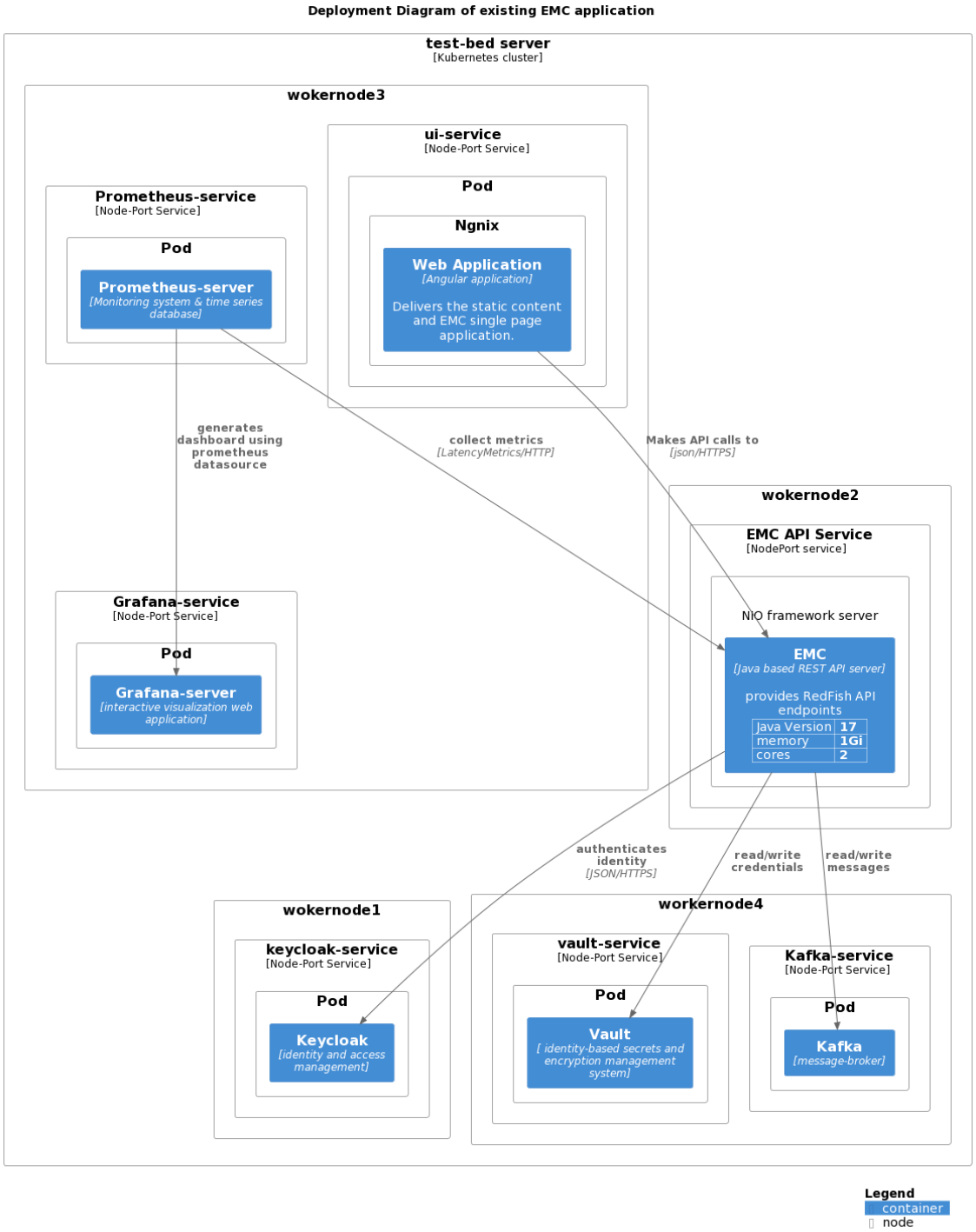


Figure 2.10 – Existing EMC application
This diagram contains only small fragment of services.

Chapter 3

Method

We started getting familar with compilation, deployment process of EMC and OMC application, tracing logs of application, creating Kubernetes pod [15], nodePort Service, and persistent volume and interacting with EMC API server and OMC API server.

Initially, We analyzed source code of EMC application and OMC application, study section of code that enforces coarsed-grained authorization and fine-grained authorization logic. We made changes in source code to disconnect old authorization logic with our new classes.

In case of EMC application, We also refer internal technical documentation of EMC application to get overview of REST API of EMC. In EMC application, Access control list (ACL) was defined in JSON file. We started writing rego file that can consume JSON and make decision based on structure of JSON. Since, EMC application is developed using Java so there are possible three ways of integration either using Wasm binary or consuming REST API via (HTTP or unix domain socket) or using Intermediate Representation file based evaluator.

Depending upon integratation approaches either HTTP based communication or Unix Domain Socket, the new classes abstracts aways low level details involved with getting decision from OPA engine but with use of Wasm binary, system get decision from Wasm runtime and with use of Intermediate Files, system get decision from Jarl evaluator. During policy evalaution, objects creations are expensive. So to increase performance, new classes contains logic that will store reference of objects. We create

objects during application initialization and store those references using singleton pattern. In the case of Jarl library, library is not published to public maven central repository. So, we need to build source code locally and add jars into EMC application with the help of Maven. Here, the role of PEP will be fufilled by EMC application and PDP will be fufilled by Open policy agent.

In case of OMC application, we didn't have (Access Control List)ACL file. So, We need to rely on REST API specification source code based on OpenAPI Specification and source code of each controller to create rego file. The source code of each OMC controller contains roles that was used for authorization and OpenAPI Specification to know REST API behaviour. Once, we had rego file, we can produce bundles with optimization levels into rego files or Wasm binary. Since, OMC application is developed using Python so there are possible two ways of integration either using Wasm binary or exposing REST API.

Once we have correct rego files, We start with working with following approaches. OPA contains all logic used by EMC and OMC application to make decisions. OPA takes roles, resource, action as input and process against rego to produce output like boolean or set of resources that are allowed.

3.0.1 First Approach: Centralized PDP

In this approach, OPA engine is deployed as REST API server into separate node which is not used by EMC application. OPA engine exposes REST API endpoints that makes possible for EMC application to interact with OPA to get decision when roles, resources, action is provided.

1. In the first phase, we use rego language to express coarsed-grained authorization and fine-grained authorization logic. Rego file uses JSON file to find permissions of endpoints and roles that contains permission. This helped us to understand the expressiveness of the Rego. Rego file used during experiment is shown in following listing 3.0.1.0.0.1 and 3.0.1.0.0.2.

```
package authz.redfish.v1.policy

import data.Roles as emc_roles
import data.Statements as emc_policies
import future.keywords

default allow =false

allow if {
   policy = emc_policies[_]
   regex.match(policy.Resource, input.resource)
   policy.Method == input.method
   some index, role in input.roles
   some perm in emc_roles[role]
   perm == policy.Permission
}
```

Listing 3.0.1.0.0.1 – coarsed-grained authorization logic

```
      package authz.redfish.v1.fine.policy

      import data.Roles as emc_roles
      import data.Statements as emc_policies
      import future.keywords

      default allow =false

      allow if {
         policy = emc_policies[_]
         regex.match(policy.Resource,
            ↪ input.resource)
         policy.Method == input.method
         some index, role in input.roles
         some perm in emc_roles[role]
         perm == policy.Permission
      }


```

```
      batch_allow = [url |
      some i
      url := input.resources[i]
      method := input.methods[i]
      allow with input as {
            "method": method,
            "resource": url,
            "roles": input.roles,
         }
      ]
```

Listing 3.0.1.0.0.2 – fine-grained authorization logic

The rego files can be organized into package. Based on the package name, REST API endpoints are generated.
For example: <ip>:<port>/v1/data/authz/redfish/v1/policy is avaiable. In our case, authz/redfish/v1/policy are obtained from package name defined in rego file.
The listing 3.0.1.0.0.3 is input for coarsed-grained authorization to OPA engine.

```
   {
      "input": {
        "resource":
           ↪ "files/upload/updateservice/package",
        "method": "POST",
        "roles": ["SecurityAdministrator"]
      }
    }
```

Listing 3.0.1.0.0.3 – input to coarsed-grained rego

To get decision from OPA, we can use curl command as shown in listing 3.0.1.0.0.4.

```
curl -X POST -d @input1.json
   ↪ http://localhost:8181/v1/data/authz/redfish/
v1/policy
```

Listing 3.0.1.0.0.4 – curl command to OPA REST API server

Based on logic defined in rego file, The output from OPA will be JSON as showin in listing 3.0.1.0.0.5 that contains value of allow. This output is suitable for coarse-grained authorization.

```
{"result":{"allow":true}}
```

Listing 3.0.1.0.0.5 – output of OPA engine

The listing 3.0.1.0.0.6 is input for fine-grained authorization to OPA engine.

```
{
"method": "GET",
"resources": ["/TaskService/Tasks/1/",
"files/upload/updateservice/package",
 "/TaskService/Tasks/3/"],
"roles": ["OmcEquipmentObserver",
"CreateJob","DeleteJob","OmcEquipmentAdministrator"
 ]
}
```

Listing 3.0.1.0.0.6 – input to fine-grained rego

Rego can desinged in such a way that the output can also provide additional data other than true or false as shown in listing 3.0.1.0.0.7. This output is suitable for post filtering processing or fine-grained authorization.

```
   {
      "allow": [
         "/TaskService/Tasks/1",
         "/TaskService/Tasks/3",
         "files/upload/updateservice/package",
      ]
   }
```

Listing 3.0.1.0.0.7 – output to fine-grained rego

2. In the Second phase, We deploy OPA engine as a standalone server into Kubernetes cluster. In our testing-environment, Kubernetes can deploy pod into any of six different nodes. We create Node-Port service on OPA pod to make it accessible using ip:port. OPA exposes

a set of REST APIs that allows Java application(PEP) to connect and enforce authorization decision. All our policy files are store into Git repository. During initialization of OPA Server, init container initialize volumes that is used by OPA server. OPA server stores policy in memory. So, If policy is updated then we need to restart the OPA server again with new policy files or we can use REST API endpoints to update policy.

3. In the third phase, we made changes to the implementation of the existing PEP. In existing application, there is class called PermissionHandler that contains logic to make decision based on policy expressed in JSON files. PermissionHandler class takes input like roles of user, resources that user wants to access and evaluates decision. Right after authentication from KeyCloack, we get JWT token that contains useful information like roles of user. AuthorizationFilter has method that can decode encrypted JWT token. We will swap the PermissionHandler class with our OPARestAuthz class. OPARestAuthz class encapsulates all details needed to get decisions from OPA Server. This will help to integrate service with OPA. Depending upon how services are built, each implementation of PEP will vary.

4. After integrating, OPARestAuthz will create appropriate (serialize object) JSON and forward JSON to OPA REST API endpoints to get decision and OPARestAuthz will deserialize JSON into boolean. PEP will enforce the basis of the result of the decision. In the process of authorization, PEP will collect relevant data from the JWT token and will query PDP along with all relevant attributes to get a decision.

5. In the fourth phase, we will evaluate the performance of our solution using Prometheus and Grafana. We will generate metrics like counter to record the number of correct and incorrect decisions made by OPA, Summary metrics to measure the latency of decision made with use of new implementation into system. Grafana can be used to generate dashboard that contains graphs.

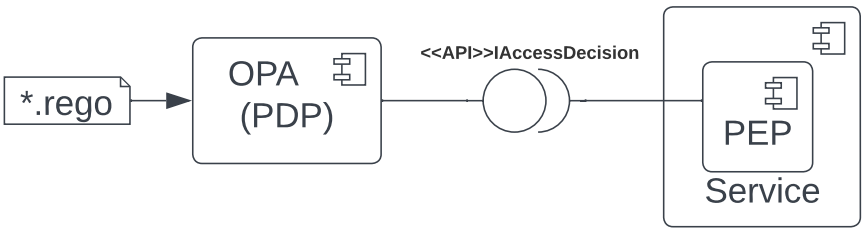


Figure 3.1 – (High-level) Integration of OPA with existing service

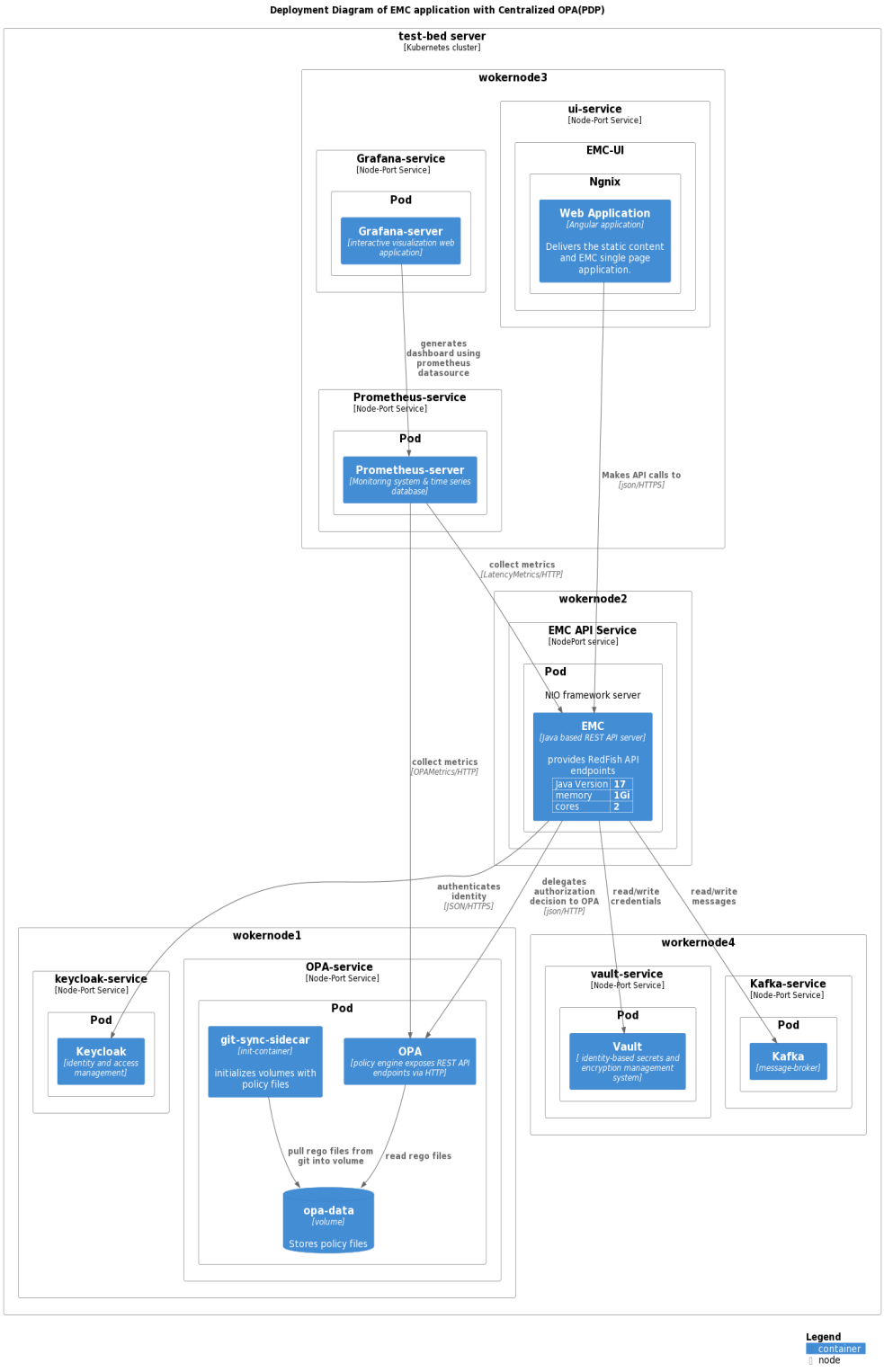


Figure 3.2 – Deployment diagram of EMC integratation with cental OPA
This diagram uses C4 model notations.
The diagram contains only small part of application.

3.1 Second Approach: Distributed PDP

In this case, we will bring PDP close to PEP i.e where EMC application resides. It is possbile by use of library called Jarl or using Wasm binary inside JVM or deploying OPA as a sidecar that opens Unix Domain socket. In all the approaches of Distributed PDP, during deployment of EMC application, init-container named git-sync-container pull IR files, Wasm binary bundles, normal bundle from git to filesystems of container which is also mounted to filesystems that was used by EMC application.

3.1.1 implementation based on Jarl library

1. we used previous rego files that was used in OPA server. we compile rego files into Intermediate Representation (IR) file. It represents a query plan, steps needed to take to evaluate a query (with policies). The plan format is a JSON encoding of the intermediate representation (IR) used for compiling queries and policies into Wasm binary. Since object creation from IR files are expensive, so during initialization of EMC application, objects are created based on IR files and references are stored.

2. The new class abstracts all low-level details for interacting with Jarl evaluator and uses Jarl library[16].

3. After integrating, we will provide input in the form of JSON to Our new class to get decisions. PEP will enforce the basis of the result of the decision.

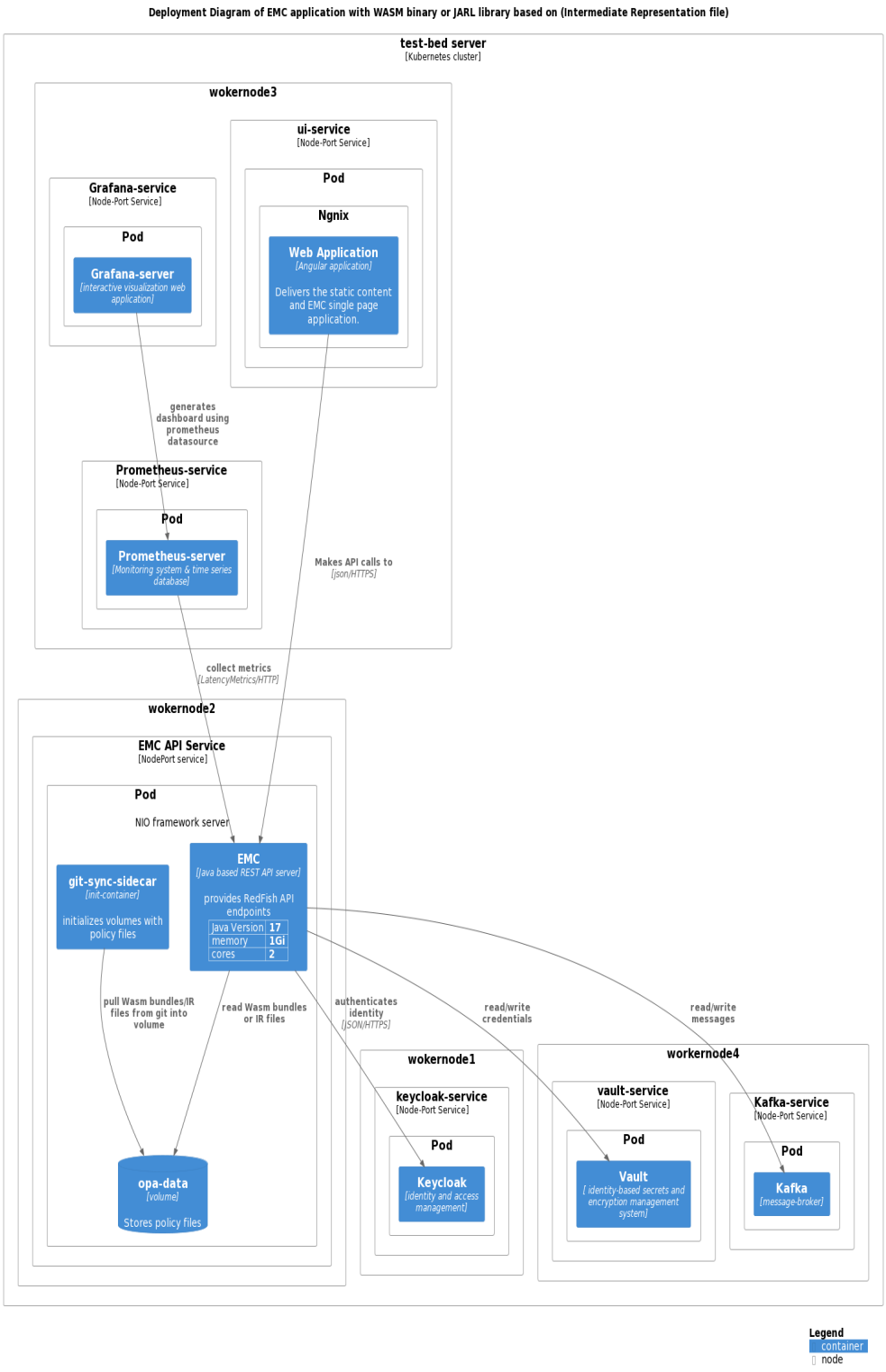


Figure 3.3 – Deployment diagram of implementation based on Wasm binary or Jarl library

3.1.2 implementation based on Wasm binary

1. we compile rego files into bundle containing Wasm binary. Since object creation from Wasm bundles are expensive, so during initialization of EMC application, objects are created based on Wasm binary and references are stored.

2. The new class uses Wasm library[17] and abstracts all low-level details for interacting with Wasm runtime.

3. After integrating, we will provide input in the form of JSON to Our new class to get decisions. PEP will enforce the basis of the result of the decision.

3.1.3 implementation based on OPA engine as a SideCar

1. we compile Rego policies into bundle containing rego.

2. The new class uses library called reactor-netty[18] and abstracts all low-level details for interacting with OPA engine via Unix domain socket.

3. After integrating, we will provide input in the form of JSON to Our new class to get decisions. PEP will enforce the basis of the result of the decision.

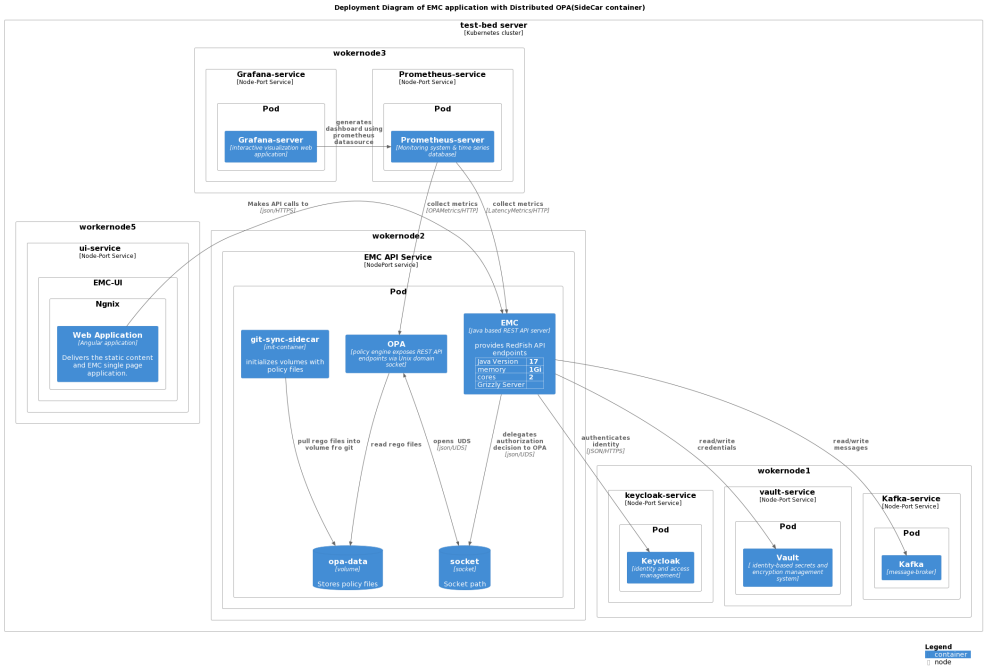


Figure 3.4 – Deployment diagram of EMC integratation with SideCar container OPA

3.2 Post processing within controller

The listing 3.0.1.0.0.7 is list of string that are allowed to display in the response. All the existing REST API controllers that uses fine-grained authorization logic creates response based on Java objects that contains allowed string along with other meta-data. So we need to create new List of objects that is allowed to be present based on output of fine-grained authorization.

```
1    Authz authz= WASMAuthz.getInstance();
2    List<String> filterUris = authz.isAllowed(uris,
        ↪ methods, roles);
3    Set<String> filterSet =
        ↪ filterUris.stream().collect(Collectors.toSet());
4
5    var filteredTasks=taskStore.getAll().stream()
6                 .filter(task ->
                       ↪ filterSet.contains(task.getPayloadTargetUri()))
7                 .map(Task::asReference)
```

```
8                   .collect(Collectors.toList());
```

Listing 3.2.0.0.0.1 – post-processing

3.3 Assessing reliability and validity of implementation

We used multiple ways to test correctness of our implementation.

- In first approach, We did manual testing by executing curl operations against protected REST api endpoints with underprivileged role, we compared actual respose with expected response. Similary, We executed curl operations against protected REST api endpoints with privileged role, comparison of actual respose with expected response was made.
- In second approach, We kept PermissionHandler and new class together in our service. We compare the result of new class with result of PermissionHandler. `correct_decisions` metrics are incremented whenever decisions are indentical and `incorrect_decisions` metrics are incremented whenever decisions are different. Prometheus server collects those metrics. Grafana was used to generate graphs. This approach is tedious since we need to call every endpoints to know how far we have testcoverage.
- In final stage, After integrating with OPA, we executed all old Functional test. It passed all test case that was used within authorization. This testing covers almost all of the endpoints and usecases.

Chapter 4

Measurements of performance

Mulitple namespaces are created where each namespace contains deployments of different implementation of EMC application. Those EMC applications have different NodePort service.

During experiment, we executed same REST api of all version of EMC applications at the same time. Similary, we did same approach with other endpoints. For every invocation of REST api, each EMC application hosted on different namespaces generates prometheus metrics which are later collected by central Prometheus server.

Latency is measured by generating Summary metrics on ContainerRequestFilter(Prefilter) of Grizzly server and Summary metric is reported back to registry before exiting ContainerRequestFilter(PostFilter) of Grizzly server. This time measures the time taken from request creation to response creation. This timings are unaffected by position of client, or network bandwidth of client. Grafana was used to generate graphs from data collected on central prometheus server.

Chapter 5

Results and Analysis

5.0.1 System Performance with coarsed-grained authorization

endpoints	old Authz	OPA via HTTP	OPA via UDS	Jarl	Wasm	UJarl
ReadInternalAlarams	**0.00357**	0.00567	**0.00396**	0.0181	0.0144	0.0182
ReadRelays	**0.00306**	0.00558	**0.00394**	0.0180	0.0141	0.0159

Table 5.1 – table contains times taken from Prefilter (i.e recieving HTTP request) to PostFilter (i.e sending HTTP response)
system based on OPA via UDS is **1.10** times slower than system based old-Authz when position of rule is last.
system based on OPA via UDS is **1.28** times slower than system based old-Authz when position of rule is 4th position. Rego uses glob pattern.
old Authz refers to system that uses old authorization logic.
OPA via HTTP refers to system that uses OPA engine via HTTP
OPA via UDS refers to system that uses OPA engine via Unix Domain socket
Jarl refers to system that uses Jarl with optimized version of planevaluator
Wasm refers to system that uses Wasm binary.
UJarl refers to system that uses Jarl evaluator with unoptimized version of plan

5.1 System Performance with fine-grained authorization

tasks	old Authz	OPA via HTTP	OPA via UDS	Wasm	UJarl	comment
3	**0.00443**	0.111	**0.00826**	0.0292	1.11	old authz based system is **1.86** times slower than OPA via UDS based system
203	**0.0696**	0.0746	0.0691	**0.0684**	t/o	old authz based system is **1.01** times slower than Wasm based system
403	**0.130**	0.130	0.123	**0.106**	t/o	old authz based system is **1.22** times slower than Wasm based system
603	**0.188**	0.186	0.18	**0.145**	t/o	old authz based system is **1.29** times slower than Wasm based system
803	**0.23**	0.234	0.227	**0.18**	t/o	old authz based system is **1.27** times slower than Wasm based system
1003	**0.28**	0.287	0.276	**0.214**	t/o	old authz based system is **1.30** times slower than Wasm based system

Table 5.2 – table contains times taken from Prefilter (i.e recieving HTTP request) to PostFilter (i.e sending HTTP response) for readTasks endPoint. Rego file uses regex pattern. Wasm and Ujarl uses unoptimized version of rego. But OPA engine via HTTP and OPA engine via UDS uses optimized version of rego.
t/o represents time out.

Chapter 6

STRIDE-based threat analysis

Security Classification			
Asset	Confidentiality	Integrity	Availability
EMC	M	VH	H
OPA	L	VH	VH
Git repo	L	VH	VH
policy files	L	VH	VH
EMC-TLS Certificate	NA	VH	H
OPA-TLS Certificate	NA	VH	VH

Table 6.1 – Low (L)= Loss of confidentiality, integrity or availability is expected to have NEGLIGIBLE adverse impact on the product
Medium (M)= Loss of confidentiality, integrity or availability is expected to have MINOR adverse impact on the product
High (H)= Loss of confidentiality, integrity or availability is expected to have MAJOR adverse impact on the product
Very High (VH) Loss of confidentiality, integrity or availability is expected to have SEVERE adverse impact on the product
N/A= Loss of confidentiality, integrity or availability is expected to have NO adverse impact on the product

Thread modeling diagram of Central OPA integratation with EMC

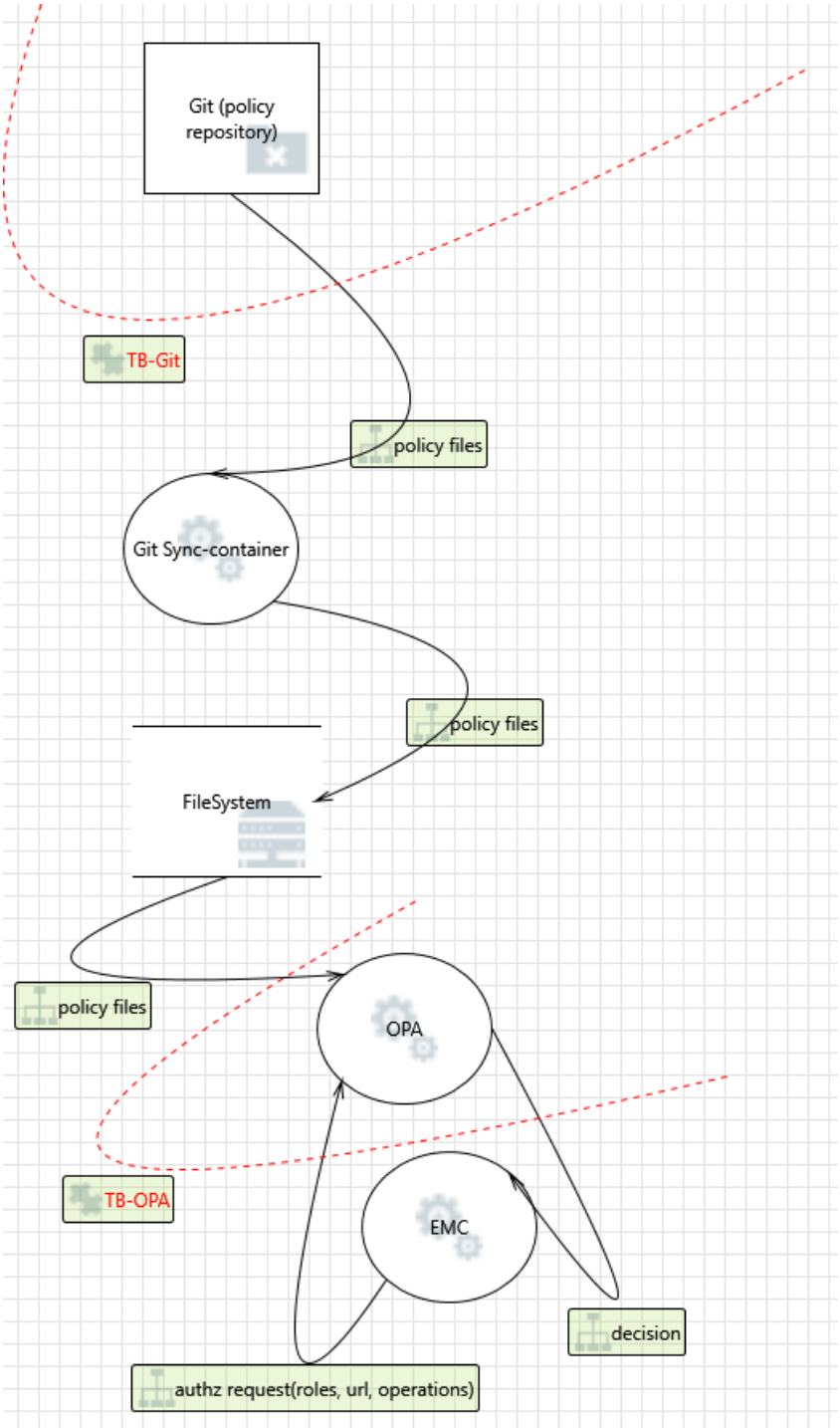


Figure 6.1 – Threat Modeling diagram of Central OPA integration with EMC

list of potential Threads within TB-OPA trust bound using STRIDE model

Trust Bound	OPA		Data Flow		EMC	
TB-OPA	Exisitng Mechanisms	Vulnerabilities	Exisitng Mechanisms	Vulnerabilities	Exisitng Mechanisms	Vulnerabilities
S	token-based and TLS-based authentication on OPA server	-	-	-	mTLS based authenticatiosn	-
T	-	it is open-source code. we don't have control	HTTPS	-	emc images are pulled from secured registry and runnning containers are protected by kubernetes security mechanism	-

R	logger exists and are displayed into console.	loggers are displayed in console. logs disappears when pod destroyed or crashed that might difficult to track activity. We need to workaround to store logs into disk using splunk	-	-	General logging in EMC and security loggings	-
I	masking of sensitive data is possible	-	HTTPS	-	Logger can filter out sensitive data	-

D	only valid tokens can invoke REST api operations. OPA official image go through vulnerablity scans in every release phase.	-	-	Access control of EMC is taken care by OPA, only decisions of OPA are enforced by EMC	-	-
E	OPA provides a lot of concurrent routines (as long as underlying node can provide cores) that can handle multiple requests. kubernetes take care of health of pod object if pod becomes unresponsive.	OPA is not running HA setup. If OPA has vulnerablity then dependent systems like EMC is affected.	-	-	kubernetes take care of health of Stateful object if pod becomes unresponsive.	loss of OPA will affect avaiablity of EMC application.

Table 6.2 – Thread tables within TB-OPA trust bound
S=Spoofing, T=Tampering, R=Repudiation, I=Information disclosure, D=Denial of service, E=Elevation of privilege

Thread modeling diagram of Distributed OPA integratation with EMC

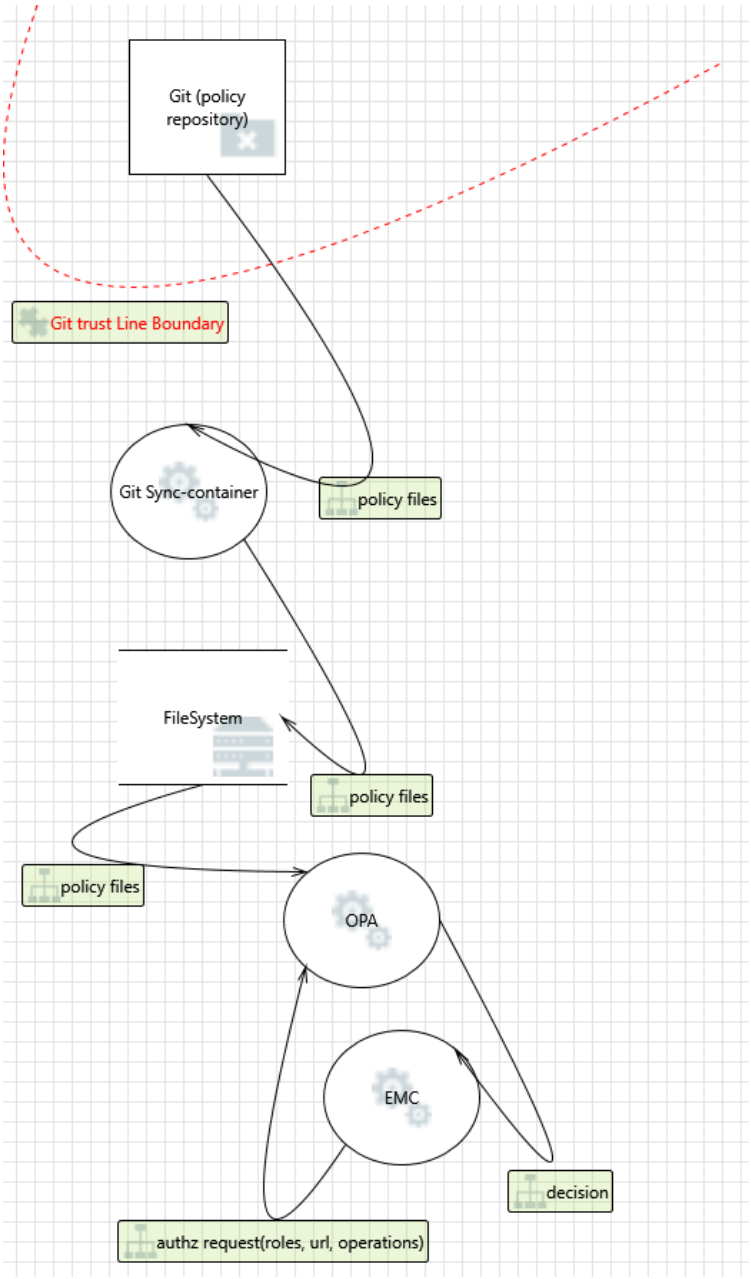


Figure 6.2 – Threat Modeling diagram of Distributed OPA integration with EMC. The diagram will be similar for Jarl, Wasm based implementation, or OPA as sidecar container.

list of potential Threads within TB-Git trust bound using STRIDE model

Trust Bound	Git		Data Flow		EMC	
TB-Git	Exisitng Mechanisms	Vulnerabilities	Exisitng Mechanisms	Vulnerabilities	Exisitng Mechanisms	Vulnerabilities
S	HTTPS based cloning. Git server is authenticated by x509	-	-	-	Git client provides user credentials to authenticate.	-
T	only authorized user having write permission can update files. Git-sync container has no write permissions.	-	HTTPS	-	It is open source and vulnerability scans are made during release.	we don't have control over source code tampering.

R	all activity of Git repo can be tracked	-	-	-	only errors can be logged into disk.	logs disappears after pod initializes that might difficult to track activity. We need to workaround to store logs into disk using splunk
I	policy files doesn't contains any information of user. Policy files is defined interms of roles.	-	-	-	there is no any senstive information	-

D	Git can be configured to be HA	loss of git avaiablity will affect newly deployed OPA service	-	adequate size of (1GB) volumes are defined for container while size of policy files is about 120 MB	-	-
E	Git can be configured with mulitple users that contains different permission	-	-	-	process of container is executed by root.	-

Table 6.3 – Thread tables within TB-Git trust bound
S=Spoofing, T=Tampering, R=Repudiation, I=Information disclosure, D=Denial of service, E=Elevation of privilege

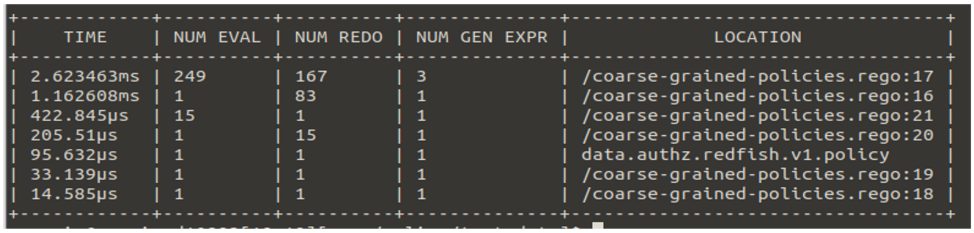
Chapter 7

Discussion

Performance depends upon evaluatation time taken by authorization logic and system integration performance (like communication, serialization/ deserilization overhead).

Policy evaluation performance

Intially, our rego file was designed to consume external json file containing ACL policy. During rule evaluation of our unoptimized rego file 7.0.0.0.0.1, OPA engine needs to iterate through each entry of Statements untill it matches with input.resource. If the match is found then, OPA again needs to iterate each entry of Roles to match with permissions associated with resources. So, the evaluatation time can be represented as $\mathcal{O}(size(Statements) * size(roles))$. In this case, the time for evaluation of policy by OPA is dependent on position of entry within json file and performances suffers with growth of size of json file. But if we create bundle from rego file and json file, entry of json are order



TIME	NUM EVAL	NUM REDO	NUM GEN EXPR	LOCATION
2.623463ms	249	167	3	/coarse-grained-policies.rego:17
1.162608ms	1	83	1	/coarse-grained-policies.rego:16
422.845µs	15	1	1	/coarse-grained-policies.rego:21
205.51µs	1	15	1	/coarse-grained-policies.rego:20
95.632µs	1	1	1	data.authz.redfish.v1.policy
33.139µs	1	1	1	/coarse-grained-policies.rego:19
14.585µs	1	1	1	/coarse-grained-policies.rego:18

Figure 7.1 – profiling of rego without any optimization

```
package authz.redfish.v1.policy

import data.Roles as emc_roles
import data.Statements as emc_policies
import future.keywords

default allow =false

# METADATA
# title: authorize
# description: A rule that determines if input is
   ↪ allowed.
# authors:
# - Prashanna Rai <prai931024@gmail.com>
# entrypoint: true
allow if {
   policy = emc_policies[_]
   glob.match(policy.Resource, ["/"],input.resource)
   policy.Method == input.method
   some index, role in input.roles
   some perm in emc_roles[role]
   perm == policy.Permission
}
```

Listing 7.0.0.0.0.1 – Small fragment of rego file that was generated using optimization=0

```

"Roles": {
   "CreateJob": [
      "createAggregationSource",
      "patchAggregationSource",
      "readAggregationService",
      "readPublicKey",
      "readTaskService"
   ],
   "DeleteJob": [
      "deleteAggregationSource",
      "readAggregationService",
```

```
            "readPublicKey",
            "readTaskService"
        ]
    },
    "Statements": [
        {
            "Method": "PATCH",
            "Permission": "patchAggregationSource",
            "Resource":
                ↪ "AggregationService/AggregationSources/*"
        },
        {
            "Method": "POST",
            "Permission": "updateSnmpCredentials",
            "Resource":
                ↪ "AggregationService/AggregationSources/*/
            Actions/Oem/
Ericsson2AggregationSource.RegenerateSNMPCredentials"
        }
    ]
```

Listing 7.0.0.0.0.2 – Small fragment of JSON file that contains ACL

with use of optimization flag 1 or 2, the rules are transformed in such a way that new set of rules are created with each entry from array. During evaluation, only lookup and comparison is only needed. Now, the evaluation time remains constant even with the growth of size of policy and is unaffected by position of rule. For example, in our case rule body contains equaltiy comparison based on role without having to compare permissions. So, the evaluatation time can be represented as $\mathcal{O}(constant)$.

```
package authz.redfish.v1.policy

default allow = false

allow {
    __local15__1 = input.resource
    glob.match("AggregationService", ["/"],
        ↪ __local15__1)
    "GET" = input.method
```

```
9       __local1__1 = input.roles[__local0__1]
10      "CreateJob" = __local1__1
11   }
12
13   allow {
14      __local15__1 = input.resource
15      glob.match("AggregationService", ["/"],
           ↪ __local15__1)
16      "GET" = input.method
17      __local1__1 = input.roles[__local0__1]
18      "DeleteJob" = __local1__1
19   }
```

Listing 7.0.0.0.0.3 – Small fragment of rego file that was generated using optimization=1

with use of optimization flag 2, the rules are further made compact. The transitive property of assignment are replaced by using copy propogation[19]. For example: rego file of 7.0.0.0.0.3 line number 9 and 10 will be replaces with single equaltiy comparison. So the new rego of line nunber 8 will have only equaltiy comparison as shown in listing 7.0.0.0.0.5.

TIME	NUM EVAL	NUM REDO	NUM GEN EXPR	LOCATION
533.65µs	1	1	1	data.authz.redfish.v1.policy
91.513µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:6
58.591µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:834
56.18µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:180
47.725µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:594
47.425µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:234
46.64µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:534
45.19µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:114
45.179µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:648
42.187µs	2	1	2	/optimized/authz/redfish/v1/policy.rego:576

Figure 7.2 – profiling of rego with optimization

```
1   package authz.redfish.v1.policy
2
3   default allow = false
4
5   allow {
6      glob.match("AggregationService", ["/"],
          ↪ input.resource)
7      "GET" = input.method
```

```
8       "CreateJob" = input.roles[__local0__1]
9    }
10
11   allow {
12      glob.match("AggregationService", ["/"],
           ↪ input.resource)
13      "GET" = input.method
14      "DeleteJob" = input.roles[__local0__1]
15   }
```

Listing 7.0.0.0.4 – Small fragment of rego file that was generated using optimization=2

we have total of 168 rules in EMC application and 68 different permissions for different endpoints. In case of an old Authorization systems, the position of ReadInternalAlarams rule in HashMap entry is 167^{th}. The position of ReadRelays rule in HashMap entry is 4^{th}. Upon inspection of authorization code 7, it is found that implementation relies on linear search for finding permission based on resource and method. It takes almost 1 lookup from role. So, the evaluation time of PermissionHandler can be represented in $\mathcal{O}(n)$. Here, the value of n is ≤ 143.

```
 public Optional<String> getPermission(final String
    ↪ key) {
 final var _key = key + (key.endsWith("/") ? "" : "/");

   return map != null ? map.entrySet().stream()
         .filter(x -> _key.matches(x.getKey()))
         .map(Map.Entry::getValue).findFirst() :
         Optional.empty();
}

public boolean isAllowed(@Nonnull List<String>
   ↪ tokenRoles, @Nonnull String requestKey) {
 var allowed = false;
 var opt = policyMap.getPermission(requestKey);

 if (opt.isPresent()) {
    allowed = rolesMap.getRoles(opt.get()).stream()
       .anyMatch(tokenRoles::contains);
```

```
  }
  return allowed;
}
```

So, the implementation of old systems is affected by position of rule in HashMap entry i.e it uses early-exit strategy to evaluate input against policy i.e it traverses in HashMap entry to find rule and then evaluate against that rule.

System integratation performance

In table 5.1, we can see that times taken by old Authorization system grows linearly when we make evaluatation of rule which is in last position.

However, In table 5.1, we can see that time taken by system based on OPA via UDS for evaluating ReadInternalAlarams and ReadRelays endpoints is almost same. So ,evaluatation is unaffected by position of rule. The system based on OPA via UDS is **1.10** times slower than system based old-Authz when position of rule is last. system based on OPA via UDS is **1.28** times slower than system based old-Authz when position of rule is 4^{th} position. Here, Rego file uses glob pattern for pattern matching, rule-indexing[20] kicks in. OPA engine internally creates Trie data-structure[13] based on all the equality expression i.e(resource,method,roles) that enables faster lookup. So, the position of rule doesn't affect evaluation time. Also, the evaluation time reduces to large extend.

But in the table 5.2, rego file uses regex pattern for pattern matching. OPA engine internally creates Trie data-structure for method, roles only. During rule evaluation, position of rule doesn't have affect if lookup from Trie data-structure returns only one but if there are more than one rules, then position has impact to size of rules returned from Trie lookup.

In case of using OPA engine as a server with bundles generated from optimization flag =2. Rego files are transformed into new rego files which doesn't require iteration. During rule evaluation, OPA rule engine uses rule-indexing to list candidates of rules that are likely to satisfy the input. The listing 7.0.0.0.0.5 is input used for finding explaination steps involved within OPA engine.

```
{
   "method": "GET",
   "resource": "Oem/Ericsson_2/Relays",
   "roles": [
      "OmcEquipmentObserver",
      "CreateJob",
      "DeleteJob",
      "OmcEquipmentAdministrator"
   ]
}
```

Listing 7.0.0.0.0.5 – input to OPA engine

In case of rule defined using regex pattern, OPA engine uses following strategy.

```

{
   "QueryID": 0,
   "ParentID": 0,
   "Locals": null,
   "LocalMetadata": {},
   "Message": "(matched 264rules, early exit)",
   "Ref": [
     {
       "type": "var",
       "value": "data"
     },
     {
       "type": "string",
       "value": "authz"
     },
     {
       "type": "string",
       "value": "redfish"
     },
     {
       "type": "string",
       "value": "v1"
     },
```

```
        {
          "type": "string",
          "value": "policy"
        },
        {
          "type": "string",
          "value": "allow"
        }
      ]
    }
```

Listing 7.0.0.0.0.6 – Small fragment of JSON file that contains explanation of evaluatation steps of OPA engine when using regex

In case of rule defined using glob pattern, OPA engine uses following strategy.

```
  {
     "QueryID": 0,
     "ParentID": 0,
     "Locals": null,
     "LocalMetadata": {},
     "Message": "(matched 2rules, early exit)",
     "Ref": [
       {
         "type": "var",
         "value": "data"
       },
       {
         "type": "string",
         "value": "authz"
       },
       {
         "type": "string",
         "value": "redfish"
       },
       {
         "type": "string",
         "value": "v1"
       },
       {
```

```
25          "type": "string",
26          "value": "policy"
27        },
28        {
29          "type": "string",
30          "value": "allow"
31        }
32      ]
33    }
```

Listing 7.0.0.0.7 – Small fragment of JSON file that contains explanation of evaluatation steps used by OPA engine when using glob

Thus, search space is reduced to large extend in case of rule with glob pattern matching. The rule evaluation is unaffected by position of rule defined in rego file only if all the equality expression can be indexed. If there are more than one rules in candidates list, then rules are evaluated from top-down approach with early-exit strategy[21] . Since, we expressed rule with glob instead of regex, OPA engine is able to create Trie data structure for all the parameters like method, roles, and url. Thus, rules evaluation faster in case of glob pattern than rules expressed with regex pattern. Also, OPA engine supports features like partial evaluation for rule containing unknown values due to which caching is done for previously evaluated rule. Due to which all of the conditions, OPA engine performs well for if number of rules are large and fine-grained authorization.

Since, intergration of OPA via UDS has some overhead, due to which some latency can be seen. But among all the alternatives of integratation to EMC application(Java based application), OPA via Unix domainSocket performs well and less overhead for coarsed-grained authorization.

Altough Jarl usage in EMC application was expected to have low latency for policy evaluation since Jarl library doesn't require REST api or unix-based socket communication to make evaluation of policy but it didn't perform as expected.

Upon investigation of Jarl library[16], we found out that Jarl evaluation engine produces the logs 7.4 at each execution steps during policy evaluation. This logs might also have affected performance to some extend. Jarl uses RE2/J library[22] for every rule containing regex pattern matching. Since

we have rules defined with regex pattern and execution time grows linearly with String size. Also, Jarl doesn't support glob pattern matching. So, we try to express rules with contains function for simple string matching of rego whenever possible. This contains [23] functions can only be used for simple string but not for pattern matching. With the use of contains fucntions also, it didn't help in performance. Also, IR files that was generated from rego files doesn't contain any information of rule indexing which is why Jarl have to execute rule from top-down order during runtime which means the position of rule will affect evaluation time. In worst, if rule that is in last position will have longer evaluation time. If Jarl had use index lookup, it would be atleast efficient to OPA server.

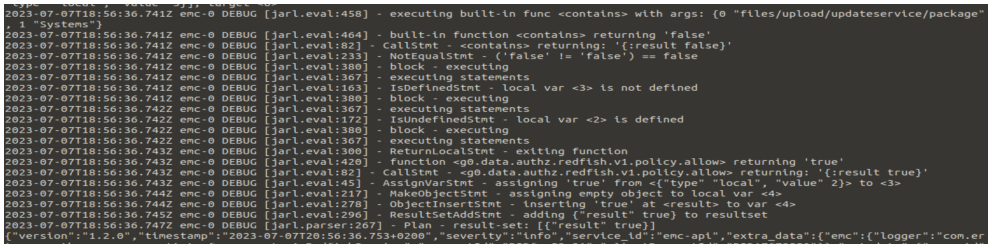


Figure 7.3 – logs of EMC application with Jarl library.

With use of Wasm binary into JVM, it was possible to get evaluation result, but the performance was degarded because the library[17] which we had use contains some overhead associated when executing Wasm binary using Wasm time runtime due to which it suffer overall performance. Java program cannot interact the Wasmtime directly, and there's an additional layer(Wasmtime-java) required [24].

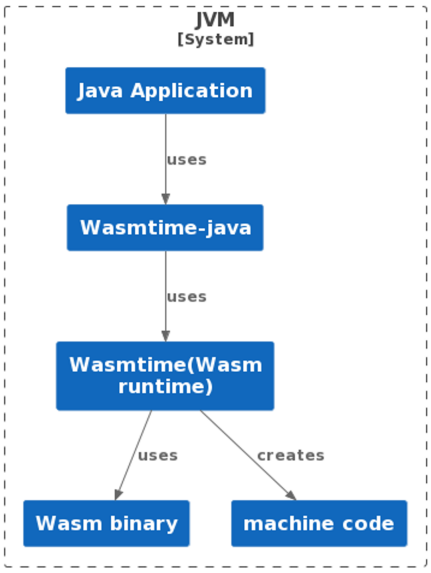


Figure 7.4 – interaction layer involved with Wasm binary

Wasm binary doesn't contains any information related to rule-indexing thus, rules are evaluated top-down approach using early-exit strategy. In the case of coarsed-grained authorization, system based on Wasm binary performs well i.e (**1.26** times faster) than system based on Jarl library but **3.7** times slower than OPA engine based on unix domain socket.

However in the case of fine-grained authorization, as data grows linearly, our old authorization logic gets slowers but Wasm based systems performs well. In fine-grained authorization, systems needs to filter out all the datas collected during REST API operations with aim that HTTP response shows data that are allowed for given roles. Our old authorization logic uses iteration for each data, it got slower with large number of iterations. But in case of Wasm based system, We provide bulk of data within single input to OPA engine or Wasm binary evaluates decisions. Wasmtime uses Just in time compilation mode[25] which might have enable loop unrolling for larger iterations.

With the use of optimized bundle to OPA server, query evaluatation time is reduced to large extend but because of comunication medium, there are some communication overhead. If we refer table 5.1 , we can see time taken by system with OPA engine via HTTP is **1.4318** times slower than OPA via

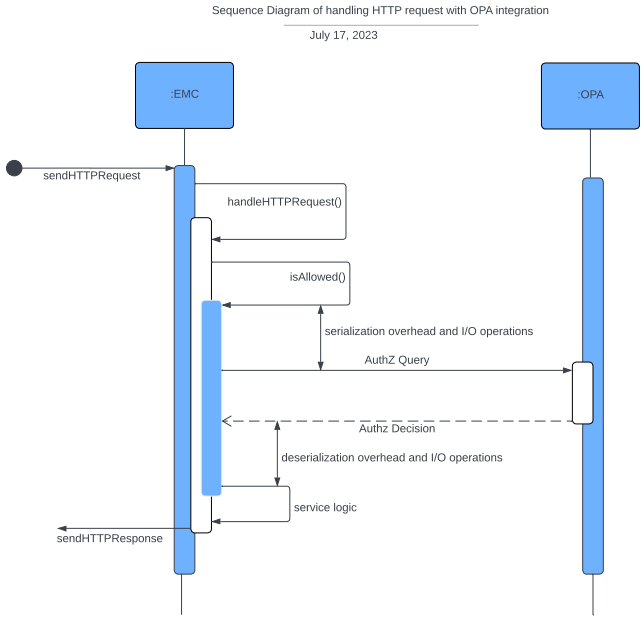


Figure 7.5 – Sequence diagram of handling HTTP request by EMC application with OPA integration. This diagram considers only high level details not classes involved in this process. This diagram will be similar for OPA via unix domain socket, Jarl library, Wasm based system.

UDS. Therefore, the overhead of communication can be reduced to large extend if we choose unix domain socket.

To sum up, with use of equality expression that are supported for rule-indexing, OPA engine is able to reduce the evaluation time. But developer still need to consider communication overhead associated each integratation approach with OPA to gain performance. In case of coarsed-grained authorization, system integration via unix domain socket has low overhead among all the approaches and there was **1.10** times slower than permissions handler. In case of fine-grained authorization, System integration with Wasm binary performs well has low overhead among all the approaches and there was **1.30** times faster than permissions handler with 1003 datas.

If we choose central OPA integration with service, deployment of new version of service will have no any impact. But if we update rego into git repository, pod containing OPA server needs to be restarted. Alternatively,

we can update rules with REST API[26] using PUT method. This method doesn't require server to be restarted.

If we choose Distributed OPA integration (based on Jarl, Wasm binary) with service, deployment of new version of service will have new policy from git repository. However, If we choose OPA integration via UDS then sidecar needs to be restarted.

Chapter 8

Conclusions and Future work

8.1 Conclusions

With help of Generic policy engine i.e (Open Policy Agent), it was possible to express all our existing access control list using rego language. Rego language being declarative language author needs to focus only providing logic. OPA can be further compiled using optimization flag to generate new set of rego files that can be evaluated in constant time. Open Policy Agent provides us configurations to run OPA as REST API server which can be exposed through unix-domain socket. OPA opens for many possible ways of integratation with any programming languges.

OPA can compile rego files to Wasm binary which then can be evaluated in any Wasm runtime. Wasm runtime that are available right now have good support within web browsers. The integratation of Wasm runtime like Wasmtime with programming language, makes possible to execute Wasm binary within any programming language system. Additionally, OPA itself ships with a Wasm runtime, which makes it possible execute bundle containing Wasm binary.

Also, OPA can compile rego files to IR files[27] which can be used to create evaluation engine. We used Jarl[28] library that uses (Intermediate Representation) IR files as input which is also evaluator that can be used as library into Java language. But this library didn't help with performance improvement since IR files didn't have any information related to rule indexing. So OPA as policy engine wins performance since OPA used rule-indexing for policy evaluation. Any implementation using rule-indxing[29]

i.e Trie data structure for rule lookup will always require low evaluatation time. But there is communication overhead involved with sending REST API request to OPA engine.

The performance of OPA depends upon structure of ACL representation and how much array lookups are made during evaluation. By changing ACL structure from array to oject (key-value pairs)[30] and avoiding use of array lookup as much as possible or use of glob matching[20] over regex patttern matching, use of optimization flag during bundle creation will speedup evaluation process.

OPA engine would have win over our existing policy evaluation code named (Permission Handler) only if size of ACL list is large and OPA engine is feed with bundle that was generated using -O=1 or -O=2 and if we choose communication medium that has low communication overhead like Unix Domain socket or gRPC.

As stated in this article [29] , It is found out that evaluation time of OPA engine remains almost same even for larger size of rules.

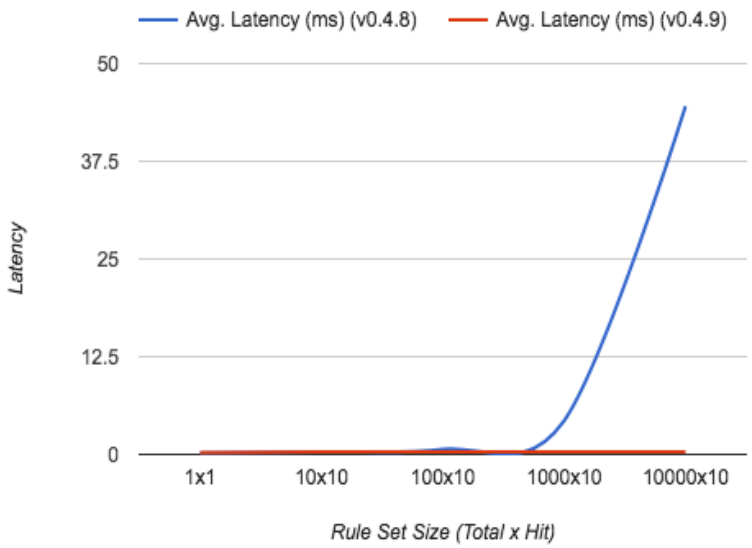


Figure 8.1 – rule-indexing performance with growth of rule size. The image is taken from blog.openpolicyagent.org

glob.match function cannot cover all possible string against pattern. For example:

- glob.match("/A/B/*", ["/"], "/A/B/C/") => false
- glob.match("/A/B/*", ["/"], "/A/B/C") => true
- glob.match("/redfish/A/B/*", ["/"], "/A/B/C") => false

But with regex.match function can cover all possible strings against single pattern. For example:

- regex.match("`/A/B/[^/]`", "/A/B/C/") => true
- regex.match("`/A/B/[^/]`", "/A/B/C") => true
- regex.match("`/A/B/[^/]`", "/redfish/A/B/C") => true

In summary, glob can cover limited number of string using pattern so we need to explicity defined all such patterns in rego rule.

With the change of business requirement, Access-control policies also changes. If rego file uses JSON file via push data approach, rule of entry should be added to ACL list. No any code change is needed to rego file. But if we are using bundle based OPA engine then OPA compiler needs to generate new bundle. In both approach, no any change is needed on microservice code.

During design of Specialized policy evaluation code, we had idea of structure of ACL So we had advantage of choosing right data structure that might supports us with faster lookup. In case of our old existing policy evaluation design code, we use HashMap that provided us faster lookup in finding permissions from role. Also, evaluating input from policy didn't need any REST API communication or executing Wasm binary i.e no any overhead for evaluation. It outperform our implementation based on Open policy Agent. But the problem with our policy evaluation code is that it is specific to our EMC application. If structure of ACL changes then we need to work with code. Also, that section of code can be only used in Java application. After implementation is developed, we need to validate, implementation performs correctly with expectation.

To sum up, our rule size were small and some overhead was involved during policy evaluatation with Open Policy Agent via REST API. So, we couldn't see performance of Open Policy agent for coarsed-grained authorization case but for fine-grained authorization, old authorization system suffer with growth of datas and implementation based on Wasm binary perform well.

Rego language is declarative language. The author needs to focus on providing correct logic for authorization system. After having correct rego, author can use different optimization level to get speed up. For example:, substituting regex pattern matching with glob for pattern matching if possible. Since, glob rules are indexed.

Author needs to use few unknowns within rego file and then use optimization flag for generating optimized rego file.

Integration approach	performance	compatibility
Go API/SDK	fastest	Go lang
Wasm	slower than OPA engine with indexed rego files but faster than OPA engine with non-indexed rego files	Wasm runtime supported languages
REST API via HTTP	slower	Any languages
REST API via UDS	faster than HTTP	Any languages
Intermediate Representation (IR)	depends upon implementation of evalutor and also IR file doesn't contain any indexed information	Any languages

Table 8.1 – OPA Integration approach

8.2 Limitations

Our implementation didn't experiment with approach for handling real time update of policy into OPA server. There is configuration to support mTLS-based authentication which I haven't experiment. I have used Bearer token into OPA servers that protects OPA server. That Bearer tokens are hardcoded into rego file which is not good practise for production. In case of central OPA server, OPA server is likely suffer from DDOS attack

so we need to investigate with rate-limit configurations. We didn't measure performance of implementation based on Wasm binary and implementation based on OPA REST API server within OMC application.

8.3 Future work

- measurements of impact on performance with implementation to OMC application.
- verfication of JWT token with private certificate.
- Open Policy Administration Layer(OPAL) [31] Integration.
- protection with TLS or mTLS certificate
- store bearer token into secrets.
- experiment with OPA engine that can be communicate through gRPC.
- investigate system performance of OPA envoy.
- experiment with OPA integration with Go-lang service using OPA sdk.
- investigatation of performance of Wasm binary execution into NodeJS application.
- investigatation with rego language with ABAC systems.

References

[1] B. Burns, *Designing Distributed Systems*. oreilly., 2018.

[2] D. Breitgand, V. Eisenberg, N. Naaman, N. Rozenbaum, and A. Weit, "Toward true cloud native nfv mano," in *2021 12th International Conference on Network of the Future (NoF)*, 2021. doi: 10.1109/NoF52522.2021.9609908 pp. 1–5.

[3] P. M. Mick Knutson, Robert Winch, *Spring Security - Third Edition*. Manning, 2017.

[4] L. Spilca, *Spring Security in action*. Manning, 2020.

[5] ——, *Spring Security in action*. Manning, 2020.

[6] "Rbac," 2020. [Online]. Available: https://csrc.nist.gov/projects/role-based-access-control

[7] "Abac," 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/journal-article/2010/adding-attributes-to-role-based-access-control

[8] styra, "Integrating opa." [Online]. Available: https://www.openpolicyagent.org/docs/latest/integration/

[9] OASIS, "xacml-semantic-model," 2013. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[10] styra, "Policy language," 2022. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-language/

[11] Opa document model. [Online]. Available: https://www.openpolicyagent.org/docs/latest/philosophy/#the-opa-document-model

[12] T. Sandall. optimization level. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-performance/#optimization-levels

[13] (2018) Trie. [Online]. Available: https://en.wikipedia.org/wiki/Trie

[14] Ericsson, "Operations manager cloud infrastructure," 2023, oMC. [Online]. Available: https://www.ericsson.com/en/portfolio/cloud-software--services/cloud-infrastructure/operations-manager-cloud-infrastructure

[15] M. Lukša, *Kubernetes in Action*. Manning, 2017.

[16] Jarl library. [Online]. Available: https://github.com/borgeby/jarl

[17] S. Lee. Opa wasm:java. [Online]. Available: https://github.com/sangkeon/java-opa-wasm

[18] reactor-netty. [Online]. Available: https://github.com/reactor/reactor-netty

[19] T. Sandall. Copy propogation. [Online]. Available: https://en.wikipedia.org/wiki/Copy_propagation

[20] indexed statements. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-performance/#use-indexed-statements

[21] early exit strategy. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-performance/#early-exit-in-rule-evaluation

[22] Re2/j. [Online]. Available: https://github.com/google/re2j

[23] *future.keywords.contains*, 2022. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-language/#futurekeywordscontains

[24] Open policy agent webassembly. [Online]. Available: https://github.com/open-policy-agent/npm-opa-wasm

[25] J. Ioannidis and G. Maguire, "Coherent File Distribution Protocol," *Internet Request for Comments*, vol. RFC 1235 (Experimental), Jun. 1991. doi: 10.17487/RFC1235. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1235.txt

[26] (2022) Create or update a policy. [Online]. Available: https://www.openpolicyagent.org/docs/latest/rest-api/#create-or-update-a-policy

[27] Intermediate representation. [Online]. Available: https://www.openpolicyagent.org/docs/latest/ir/

[28] A. Eknert. I have a plan! exploring the opa intermediate representation (ir) format. [Online]. Available: https://blog.openpolicyagent.org/i-have-a-plan-exploring-the-opa-intermediate-representation-ir-format-7319cd94b37d

[29] T. Sandall. Optimizing opa:rule indexing. [Online]. Available: https://blog.openpolicyagent.org/optimizing-opa-rule-indexing-59f03f17caf3

[30] objects over arrays. [Online]. Available: https://www.openpolicyagent.org/docs/latest/policy-performance/#use-objects-over-arrays

[31] styra, "Open policy administration layer." [Online]. Available: https://docs.permit.io/overview/what-is-permit#opa--opal