Doorman Documentation

Release 1.0.0

Mozilla

Jan 31, 2018

Contents

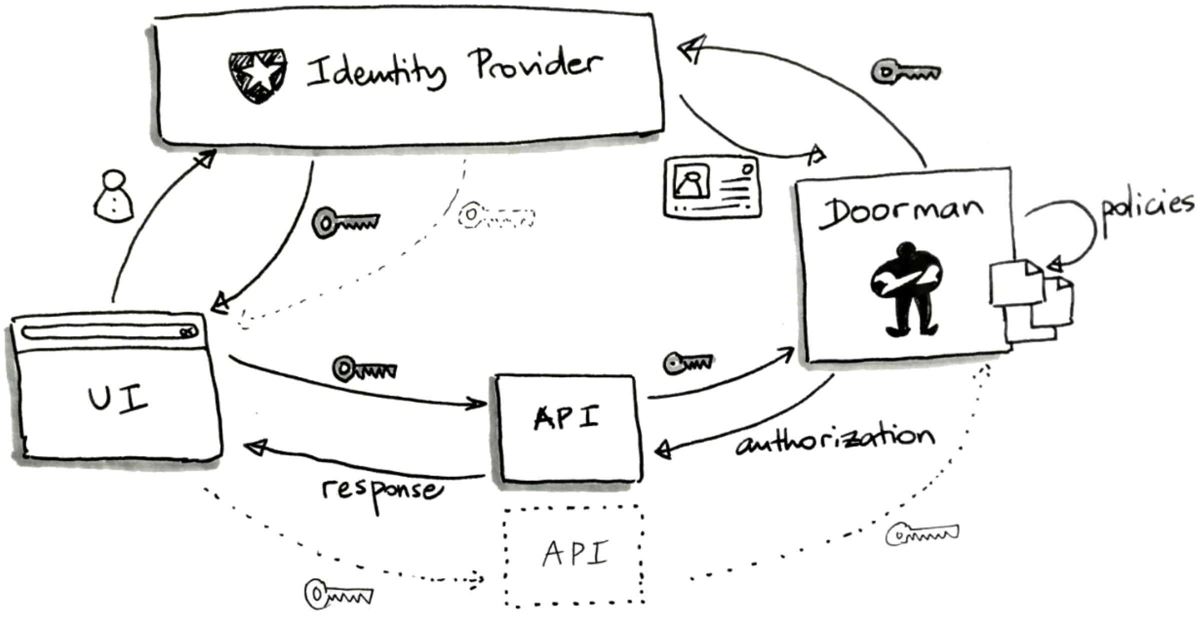
Doorman is an **authorization micro-service** that allows to checks if an arbitrary subject is allowed to perform an action on a resource, based on a set of rules (policies).

Having a centralized access control service has several advantages:

- it clearly dissociates authentication from authorization
- it provides a standard and generic permissions system to services developers
- it facilitates permissions management across services (eg. makes revocation easier)
- it allows authorizations monitoring, metrics, anomaly detection

CHAPTER 1

Workflow



It relies on OpenID Connect to authenticate requests. The policies are defined per service and loaded in memory. Authorization requests are logged out.

When a service takes advantage of *Doorman*, a typical workflow is:

1. Users obtain an access token from an Identity Provider (eg. Auth0)
2. They use it to call a service API endpoint
3. The service posts an authorization request on *Doorman* to check if the user is allowed to perform a specific action
4. *Doorman* uses the `Origin` request header to select the set of policies to match

5. *Doorman* fetches the user infos using the provided access token and builds a list of strings (*principals*) to characterize this user
6. *Doorman* matches the policies and returns if allowed or not, along with the list of principals
7. Based on the *Doorman* response, the service denies the original request or executes it

CHAPTER 2

Contents

2.1 Quickstart

2.1.1 Policies

Policies are defined in YAML files for each consuming service, locally or in remote (private) Github repos, as follow:

```yaml
service: https://api.service.org
identityProvider: https://api.auth0.com/
policies:
  - id: alice-bob-create-keys
    description: Alice and Bob can create keys
    principals:
      - userid:alice
      - userid:bob
    actions:
      - create
    resources:
      - key
    effect: allow
  -
    id: crud-articles
    description: Editors can CRUD articles
    principals:
      - role:editor
    actions:
      - create
      - read
      - delete
      - update
    resources:
      - article
    effect: allow
```

Save it to `config/api-policies.yaml` for example.

2.1.2 Run

Doorman is available as a Docker image (but can also be *ran from source*).

In order to read the local files from the container, we will mount the local `config` folder to `/config`. We'll then use `/config` as the `POLICIES` location.

```
docker run \
  -e POLICIES=/config \
  -v ./config:/config \
  -p 8000:8080 \
  --name doorman \
  mozilla/doorman
```

Doorman is now ready to respond authorization requests on *http://localhost:8080*. See *API docs*!

2.1.3 Examples

See the examples folder on Github.

2.2 Policies

Policies are defined in YAML files for each consuming service as follow:

```
service: https://service.stage.net
identityProvider: https://auth.mozilla.auth0.com/
tags:
  superusers:
    - userid:maria
    - group:admins
policies:
  -
    id: authors-superusers-delete
    description: Authors and superusers can delete articles
    principals:
      - role:author
      - tag:superusers
    actions:
      - delete
    resources:
      - article
    effect: allow
```

- **service**: the unique identifier of the service
- **identityProvider** (*optional*): when the identify provider is not empty, *Doorman* will verify the Access Token or the ID Token provided in the authorization header to authenticate the request and obtain the subject profile information (*principals*)
- **tags**: Local «groups» of principals in addition to the ones provided by the Identity Provider
- **actions**: a domain-specific string representing an action that will be defined as allowed by a principal (eg. `publish`, `signoff`, …)

- **resources**: a domain-specific string representing a resource. Preferably not a full URL to decouple from service API design (eg. *print:blackwhite:A4*, *category:homepage*, . . .).
- **effect**: Use `effect: deny` to deny explicitly. Requests that don't match any rule are denied.

2.2.1 Settings

Policies can be read locally or in remote (private) Github repos.

Settings are set via environment variables:

- `POLICIES`: space separated locations of YAML files with policies. They can be **single files**, **folders** or **Github URLs** (default: `./policies.yaml`)
- `GITHUB_TOKEN`: Github API token to be used when fetching policies files from private repositories

Note: The `Dockerfile` contains different default values, suited for production.

2.2.2 Principals

The principals is a list of prefixed strings to refer to the «user» as the combination of ids, emails, groups, roles. . .

Supported prefixes:

- `userid:`: provided by Identity Provider (IdP)
- `tag:`: local tags from policies file
- `role:`: provided in *context of authorization requests*
- `email:`: provided by IdP
- `group:`: provided by IdP

Example: `["userid:ldap|user", "email:user@corp.com", "group:Employee", "group:Admins", "role:editor"]`

2.2.3 Advanced policies rules

Regular expressions

Regular expressions begin with < and end with >.

```
principals:
  - userid:<[peter|ken]>
resources:
  - /page/<.*>
```

Note: Regular expressions are not supported in tags members definitions.

Conditions

The conditions are **optional** on policies and are used to match field values from the *authorization request context*.

The context value `remoteIP` is forced by the server.

For example:

```
policies:
  -
    description: Allow everything from dev environment
    conditions:
      env:
        type: StringEqualCondition
        options:
          equals: dev
```

There are several types of conditions:

Field comparison

- type: `StringEqualCondition`

For example, match `request.context["country"] == "catalunya"`:

```
conditions:
  country:
    type: StringEqualCondition
    options:
      equals: catalunya
```

Field pattern

- type: `StringMatchCondition`

For example, match `request.context["bucket"] ~= "blocklists-.*"`:

```
conditions:
  bucket:
    type: StringMatchCondition
    options:
      matches: blocklists-.*
```

Match principals

- type: `MatchPrincipalsCondition`

For example, allow requests where `request.context["owner"]` is in principals:

```
conditions:
  owner:
    type: MatchPrincipalsCondition
```

Note: This also works when a the context field is list (e.g. list of collaborators).

IP/Range

- type: `CIDRCondition`

For example, match `request.context["remoteIP"]` with [CIDR notation](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing#CIDR_notation):

```
conditions:
  remoteIP:
    type: CIDRCondition
    options:
      # mask 255.255.0.0
      cidr: 192.168.0.1/16
```

2.3 API

2.3.1 Summary

Basically, authorization requests are checked using **POST /allowed**.

- The `Origin` request header specifies the service to match policies from.
- The `Authorization` request header provides the OpenID *Access Token* to authenticate the request.

Request:

```
POST /allowed HTTP/1.1
Origin: https://api.service.org
Authorization: Bearer f2457yu86yikhmbh

{
  "action" : "delete",
  "resource": "articles/doorman-introduce",
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "allowed": true,
  "principals": [
    "userid:ada",
    "email:ada.lovelace@eff.org",
    "group:scientists",
    "group:history"
  ]
}
```

2.3.2 Principals

The authorization request :term:principals will be built from the user profile information like this:

- `"sub"`: `userid:{}`
- `"email"`: `email:{}` (*optional*)
- `"groups"`: `group:{}, group:{}, ...` (*optional*)

They will be matched against those specified in the policies rules to determine if the authorization request is denied or allowed.

2.3.3 Authentication

Doorman relies on OpenID to authenticate requests.

It will use the `service` and `identityProvider` fields from the service policies file to fetch the user profile information.

The `Origin` request header should match one of the services defined in the policies files.

The `Authorization` request header should contain a valid *Access Token*, prefixed with `Bearer ``. This access token must have been requested with the ``openid profile` scope for *Doorman* to be able to fetch the profile information (See Auth0 docs).

The userinfo URI endpoint is then obtained from the metadata available at `{identityProvider}/.well-known/openid-configuration`.

If the obtention of user infos is denied by the *Identity Provider*, the authorization request is obviously denied.

Using ID tokens

Doorman can verify and read user information from JWT *ID tokens*. Since the ID token payload contains the user information, it saves a roundtrip to the Identity Provider when handling authorization requests.

For this to work, the `service` value in the policies file must match the `audience` value configured on the Identity Provider — the unique identifier of the target API. For example, in Auth0 it can look like this: `SLocf7Sa1ibd5GNJMMqO539g7cKvWBOI`.

Important: When using JWT *ID tokens*, only the validity of the token will be checked. In other words, users that are revoked from the Identity Provider after their ID token was issued will still considered authenticated until the token expires.

Without authentication

If the identity provider is not configured for a service (explicit empty value), no authentication is required and the principals are posted in the authorization body.

```
POST /allowed HTTP/1.1
Origin: https://api.service.org
Authorization: Bearer f2457yu86yikhmbh

{
  "action" : "delete",
  "resource": "articles/doorman-introduce",
  "principals": [
    "userid:mickaeljfox",
    "email:mj@fox.com",
    "group:actors"
  ]
}
```

It is not especially recommended, but it can give a certain amount of flexibility when authentication is fully managed on the service.

A typical workflow in this case would be:

1. Users call the service API endpoint 1. The service authenticates the user and builds the list of principals 1. The service posts an authorization request on *Doorman* containing the list of principals to check if the user is allowed

2.3.4 Context

Authorization requests can carry additional information contain any extra information to be matched in *policies conditions*.

The values provided in the `roles` context field will expand the principals with extra `role:{}` values.

```
POST /allowed HTTP/1.1
Origin: https://api.service.org
Authorization: Bearer f2457yu86yikhmbh

{
  "action" : "delete",
  "resource": "articles/doorman-introduce",
  "context": {
    "env", "stage",
    "roles": ["editor"]
  }
}
```

2.3.5 API Endpoints

(Automatically generated from the OpenAPI specs)

POST /allowed

Check authorization request

Are those `principals` allowed to perform this `action` on this `resource` in this `context`?

With authentication enabled, the principals are either read from the Identity Provider user info endpoint or directly from the JSON Web Token payload if an ID token is provided.

Status Codes

- 400 Bad Request – Missing headers or invalid posted data.
- 401 Unauthorized – OpenID token is invalid.
- 200 OK – Return whether it is allowed or not.

Request Headers

- Origin – The service identifier (eg. `https://api.service.org`). It must match one of the known service from the policies files.
- Authorization – With OpenID enabled, a valid Access token (or JSON Web ID Token) must be provided in the `Authorization` request header. (eg. *Bearer eyJ0eXAiOiJKV1QiLCJhbG…9USXpOalEzUXpV*)

POST /__reload__

Reload the policies

Reload the policies (synchronously). This endpoint is meant to be used as a Web hook when policies files were changed upstream.

> It would be wise to limit the access to this endpoint (e.g. by IP on reverse proxy)

Status Codes

- 200 OK – Reloaded successfully.
- 500 Internal Server Error – Reload failed.

GET /__heartbeat__
Is the server working properly? What is failing?

Status Codes

- 200 OK – Server working properly
- 503 Service Unavailable – One or more subsystems failing.

GET /__lbheartbeat__
Is the server reachable?

Status Codes

- 200 OK – Server reachable

GET /__version__
Running instance version information

Status Codes

- 200 OK – Return the running instance version information

GET /__api__
Open API Specification documentation.

Status Codes

- 200 OK – Return the Open Api Specification.

GET /contribute.json
Open source contributing information

Status Codes

- 200 OK – Return open source contributing information.

2.4 Misc

2.4.1 Run from source

```
make serve -e "POLICIES=sample.yaml /etc/doorman"
```

2.4.2 Run tests

```
make test
```

2.4.3 Generate API docs

We use Sphinx, therefore the Python `virtualenv` command is required.

```
make docs
```

2.4.4 Build docker container

```
make docker-build
```

2.4.5 Advanced settings

- `PORT`: listen (default: `8080`)
- `GIN_MODE`: server mode (`release` or default `debug`)
- `LOG_LEVEL`: logging level (`fatal|error|warn|info|debug`, default: `info` with `GIN_MODE=release` else `debug`)
- `VERSION_FILE`: location of JSON file with version information (default: `./version.json`)

2.4.6 Frequently Asked Questions

Why did you do this like that?

If something puzzles you, looks bad, or is not crystal clear, we would really appreciate your feedback! Please file an issue! — yes, even if you feel uncertain :)

Why should I use Doorman?

Doorman saves you the burden of implementing a fined-grained permission system into your service. Plus, it can centralize and track authorizations accross multiple services, which makes permissions management a lot easier.

How is it different than OpenID servers (like Hydra, etc.)?

Doorman is not responsible of managing users. It relies on an Identity Provider to authenticate requests and focuses on authorization.

What is the difference with my Identity Provider authorizations?

Identity Providers may have some authorization/permissions system that allow to restrict access using user groups, audience and scopes.

This kind of access control is global for the whole service. *Doorman* provides advanced policies rules that can be matched per action, resource, or any domain specific context.

Why YAML?

Policies files are meant to be edited or at least reviewed by humans. And YAML is relatively human-friendly. Plus, YAML allows to add comments.

2.4.7 Glossary

Identity Provider An identity provider (abbreviated IdP) is a service in charge of managing identity information, and providing authentication endpoints (login forms, tokens manipulation etc.)

Access Token

Access Tokens An access token is an opaque string that is issued by the Identity Provider.

ID Token

ID Tokens The ID token is a JSON Web Token (JWT) that contains user profile information (like the user's name, email, and so forth), represented in the form of claims.

Principal

Principals In *Doorman*, the *principals* is the list of strings that characterize a user. It is built from the user information, tags from the policies file and roles from the authorization request. (see Wikipedia)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

HTTP Routing Table

Index