Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Secure Coding

Laboratoire 3

Logging and access controls

Auteurs:

Soulaymane Lamrani

Responsables:

Alexandre Duc

Nathan Séville

19 juin 2022

Logs

Je me suis essentiellement basé sur les slides du cours pour décider de quoi et comment logger. J'ai utilisé la crate `simplelog` avec une configuration qu'on trouve dans les slides.

1 Niveau de loggin

- Trace: pour afficher les entrées dans les fonctions,
- Debug: pour afficher les informations pour debugger,
- Info: pour afficher les actions qui réussissent comme prévu,
- Warn: pour afficher les actions qui ont échouées (validation d'input ou contrôle d'accès),
- Error: pour afficher les erreurs, notamment quand la connection se ferme.

Il y a aussi des informations qui ne sont délibérément pas affichées, comme le mot de passe quand celui-ci échoue, par exemple.

Je n'ai pas pu prendre le temps d'intégrer le login de Casbin dans celui de simplelog, histoire d'avoir un affichage cohérent.

Contrôle d'accès

Toujours en suivant les slides du cours, j'ai utilisé le crate Casbin pour pour gérer les contrôles d'accès.

On peut identifier 3 rôles:
- HR: qui correspondrait à admin,
- Standard user: utilisateur standard,
- Anonyme: qui correspondrait à un utilisateur non connecté.

À partir du code source de base, j'ai pu déduire les règles suivantes:

_	show_users	change_own_phone	add_user	change_phone
hr	yes	yes	yes	yes
standard_user	yes	yes	no	no
anon	yes	no	no	no

Ça ne ressemble pas tout à faire à un contrôle d'accès avec lecture et écriture, mais dans ce laboratoire, il s'agit de pouvoir accéder à des fonctions en guise de ressources, qui elles font soit de la lecture, soit de l'écriture.

Nous avons donc:

- Tout le monde peut voir les informations des utilisateurs,
- Tout le monde sauf Anonyme peut changer son propre numéro de téléphone,
- Seul HR peut créer un utilisateur et changer un numéro de téléphone.

1 Type de contrôle d'accès

Comme il s'agit d'un contrôle d'accès avec des rôles, j'ai mis en place un RBAC. Cependant le problème étant que Casbin n'a pas d'adaptateur pour Rustbreak, il faut donc l'implémenter soi-même ou changer de méthode de stockage.

J'ai fait une première tentative en utilisant la crate `csv` de Rust, mais le problème est que dans le fichier `policy.csv` , pas toutes les entrées ont le même nombre d'éléments, donc à la lecture de celui-ci, il y a une erreur. Et si je veux simplement écrire avec, ça n'ajoute pas simplement à la fin, mais ça supprime le contenu pour rajouter seulement la nouvelle entrée. Donc peu probant.

Une deuxième tentative avec `FileAdapter` de Casbin, mais malheureusement, je n'arrive pas à m'en sortir avec juste le nom des méthodes et leurs arguments[1]. On pourrait tout à fait s'inspirer des adaptateurs existants, mais pour la portée de ce laboratoire, le temps investi n'en vaudrait pas la chandelle[2].

2 Implémentation

Tout se trouve dans le sous-dossier `access_control` dans le dossier du serveur.

Il y a un enum `AccessObject` qui va correspondre à la ressource à laquelle on veut accéder.

Il y a un rôle *Anon* qui a été rajouter dans `UserRole` .

Un struct `AccessControler` qui s'occupera de charger le modèle et la policy, d'activer le logging, puis de tester si une requête à pu aboutir ou non.

Et un struct `Request` qui remplacera le tuple de requête.

[1] Pour pas dire que la documentation est à peu près inexistante...
[2] J'écris ça un peu tard, veuillez m'excuser.

2.1 Modèle

Le modèle à la forme suivante

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _
g2 = _, _

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && g2(r.obj, p.obj) && r.act == p.act
```

J'ai passé bien entre 2h et 3h pour comprendre pourquoi une requête d'un HR voyait être refusée pour voir qu'il s'agissait de la dernière ligne avec `g2(r.obj, p.obj)` et non pas un problème dans la syntaxe de la requête.

2.2 Policy

Il a cette forme:

```
g2, show_users, anon

g2, change_own_phone, standard

g2, change_phone, admin
g2, add_user, admin

g, standard_user, anon
g, hr, standard_user

p, anon, anon, access
p, standard_user, standard, access
p, hr, admin, access

g, default_user, standard_user
g, default_hr, hr
```

On peut voir tout en haut, la ressource accessible par un compte anonyme. Ensuite la ressource accessible par un compte standard. Puis les ressources accessibles pas un admin (qui sera un compte HR).

Après on définit la relation entre les rôles:

- standard_user ≥ anon
- hr ≥ standard_user

On définit ensuite les policies et à la toute fin du fichier, on met les utilisateurs dans les rôles qui les correspondent. C'est ici qu'il faudra ajouter une nouvelle ligne à l'ajout d'un nouvel utilisateur.

2.3 Tests

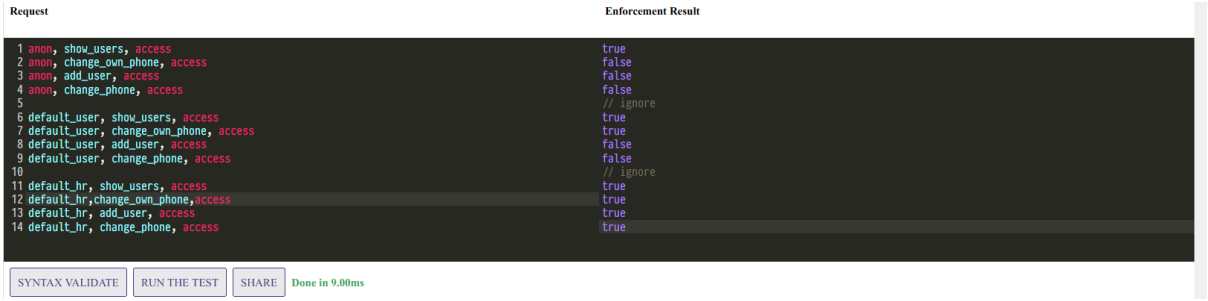

Figure 1 RBAC tests

Pour contrôler les accès, j'ai fait les différentes requêtes avec les 3 types de rôles et on peut voir que le comportement est celui attendu.

Validation d'input

Pour ceci, j'ai utilisé exactement les même regex que dans le labo précédent, mais en découplant cette fois-ci la validation de l'implémentation du trait `FromStr` demandé par `input_reader` .

La validation peut ainsi être faites côté client et côté serveur. Cela dit, si l'utilisateur passe par le client, la validation ne devrait pas être différentes entre les deux.

Problèmes connus

Quand un utilisateur anonyme tente d'accéder au ressource qui nécessite le rôle HR, à la fin des inputs, l'application client crash car un message d'erreur n'est pas correctement envoyé au client. Je n'ai pas pu me pencher de manière extensive sur ce problème.

Quand un utilisateur HR ajoute un utilisateur, celui-ci ne verra pas de policies écrites pour lui, c'est parce qu'il faut écrire un adaptateur pour Rustbreak ou un `FileAdapter` de Casbin.