HEIG-VD — SLH

Laboratoire 3 – Rapport

Loïc Herman

8 janvier 2025

1 Questions

> Voyez-vous des problèmes avec la politique spécifiée dans l'énoncé ?

Il est potentiellement dangereux d'accorder aux administrateurs des droits illimités sans aucune restriction – même les administrateurs devraient avoir certaines limitations à des fins d'audit et de sécurité. Par exemple, il pourrait être intéressant d'offrir une fonctionnalité de « sudo » pour certaines actions sensibles ou qui concernent des données sensibles ce qui permettrait de mettre en place une validation par un pair.

La politique ne précise pas ce qui se passe lorsque la relation médecin-patient prend fin – il devrait y avoir des règles claires concernant l'accès aux données historiques.

Enfin, il n'est pas fait mention des protocoles d'accès d'urgence, qui sont souvent nécessaires dans les systèmes de soins de santé. Les médecins en service d'urgences pourraient avoir besoin de lire le dossier médical d'un patient sans que celui-ci doive leur accorder le droit par son propre accès.

> Parmi les politiques définies ci-dessus, la ou lesquelles serai(en)t pénibles à implémenter s'il fallait utiliser à la place d'ABAC un modèle RBAC traditionnel ?

Les politiques les plus difficiles à mettre en œuvre dans le cadre du système RBAC sont celles qui impliquent des relations contextuelles, en particulier :

- Un utilisateur accédant à son dossier personnel
- Un médecin accédant aux données de ses patients
- Un auteur accédant à ses propres rapports

Ces scénarios nécessitent des décisions basées sur les attributs, car ils dépendent de la relation entre le sujet (utilisateur) et l'objet (ressource), et pas seulement du rôle. Dans un système RBAC, il faudrait créer des combinaisons de rôles complexes ou recourir à des contrôles au niveau de l'application, ce qui rendrait le système plus compliqué et plus difficile à maintenir, sans compter que ça rend l'utilisation de casbin plus ou moins redondante.

> Que pensez-vous de l'utilisation d'un `Option<UserID>` mutable dans la structure Service pour garder trace de l'utilisateur loggué ? Comment pourrait-on changer le design pour savoir à la compilation si un utilisateur est censé être connecté ou pas ? Est-ce que cela permet d'éliminer une partie du traitement d'erreurs ?

L'utilisation d'une `Option<UserID>` mutable dans la structure du service est problématique, car elle introduit des erreurs d'exécution potentielles et rend le comportement du code moins prévisible. Une meilleure approche consisterait à utiliser le système de types de Rust pour imposer l'état de connexion au moment de la compilation. Par exemple, nous pourrions créer deux types de services distincts :

```rust
struct UnauthenticatedService { ... }
struct AuthenticatedService { user_id: UserID, ... }
```

En quoi le `UnauthenticatedService` aurait la même méthode `pub fn login(/**/) -> Result` qui retournerait par contre une nouvelle instance de `AuthenticatedService` avec le champ `user_id` renseigné.

Cette conception éliminerait les contrôles d'authentification au moment de l'exécution et les déplacerait au moment de la compilation, réduisant ainsi la nécessité de gérer les erreurs liées à l'état de l'authentification. Les méthodes nécessitant une authentification n'existeraient que sur le service authentifié, ce qui rendrait impossible leur appel sans une authentification appropriée.

Considération importante en cas de service partagé :

Le modèle ci-dessus fonctionne bien pour les applications CLI où l'instance de service gère une session utilisateur unique. Cependant, le stockage de l'état de l'utilisateur dans le service serait dangereux dans un contexte d'API où l'instance de service pourrait être partagée entre plusieurs requêtes. Il faudrait dans un tel contexte considérer :

1. L'état d'authentification de l'utilisateur doit être géré par demande, généralement à l'aide d'une session
2. Le service doit être sans état, tout contexte utilisateur étant transmis en tant que paramètre aux appels de méthode.
3. Utiliser un pattern de service qui met en valeur ces besoins en utilisant le système de types de Rust.

> Que pensez-vous de l'utilisation de la macro de dérivation automatique pour `Deserialize` pour les types de `model` ? Et pour les types de `input_validation` ?

Pour les types de `model` : L'utilisation de la dérivation automatique pour les types de modèles est généralement acceptable, car ces types ont généralement une correspondance directe avec leur forme sérialisée. Toutefois, il convient de veiller à ce que les champs sensibles soient correctement traités lors de la sérialisation.

Pour les types de `input_validation` : La dérivation automatique n'est peut-être pas idéale dans ce cas, car la validation des entrées nécessite une logique métier et des contraintes personnalisées. Elle permet donc de créer une instance d'un type supposé validé sans effectuer la validation dans l'implémentation de `TryFrom`. Il serait préférable de mettre en œuvre une logique de désérialisation personnalisée afin de garantir une validation correcte aux limites du système.

> Que pensez-vous de l'impact de l'utilisation de Casbin sur la performance de l'application ? sur l'efficacité du système de types ?

Impact sur les performances : Casbin introduit une surcharge supplémentaire due à l'évaluation de la politique au runtime. Si cet impact peut être acceptable pour les petits systèmes, il peut devenir significatif en cas de volumes de transactions élevés. Des stratégies de mise en cache devraient être envisagées pour optimiser les performances. À titre d'exemple, des tests de produits cartésiens sur toutes les combinaisons possibles d'utilisateurs, de ressources et d'actions prenaient entre 2 et 5 minutes par cas de politique évalué, malgré l'ensemble de données de taille plutôt raisonnable. Un échantillonage stratégique a donc dû être fait pour conserver un temps d'exécution raisonnable.

Impact sur le système de type : L'évaluation de la politique de Casbin se fait à l'exécution, ce qui signifie que nous perdons certaines des garanties de Rust à la compilation. Cela crée une séparation entre le système de type fort de Rust et l'application de la politique au moment de l'exécution, rendant potentiellement le système plus difficile à raisonner statiquement. Le développement de la politique casbin est aussi plus ardu étant donné qu'il n'y a pas de complétion proposée selon le contexte offert pour chaque action.

> Avez-vous d'autres remarques ?

Non.