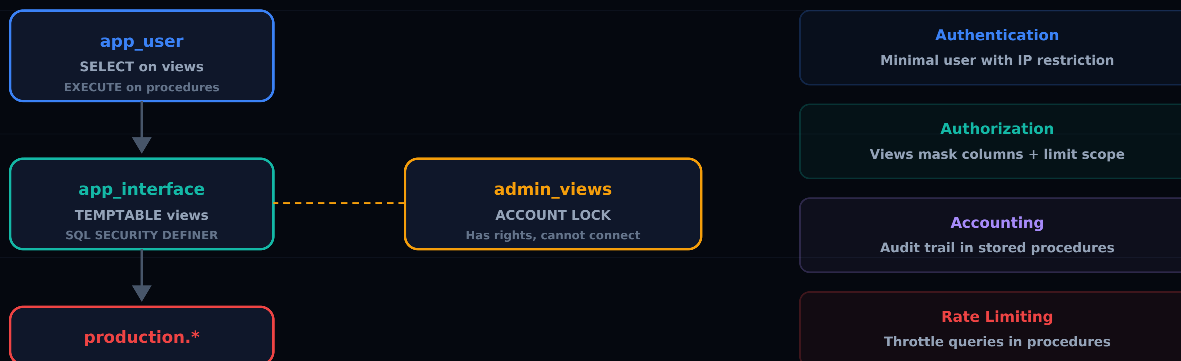
Séparons-nous physiquement

Sylvain ARBAUDIE · 4 novembre 2024

MARIADB SECURITY ACCESS-CONTROL VIEWS



Le modèle AAA appliqué aux bases de données

En sécurité informatique, le modèle AAA (Authentication, Authorization, Accounting) est un pilier fondamental. On le retrouve dans RADIUS, TACACS+, les pare-feu, les VPN... mais rarement appliqué avec rigueur aux bases de données relationnelles.

Pourtant, MariaDB / MySQL offre des mécanismes natifs qui permettent d'implémenter une véritable séparation physique entre les données sensibles et les utilisateurs applicatifs. Pas besoin d'un middleware supplémentaire, pas besoin d'un proxy coûteux. Tout est déjà là, dans le moteur.

L'idée centrale est simple : **un utilisateur applicatif ne devrait jamais avoir accès directement aux tables contenant des données sensibles**. Il devrait interagir uniquement avec des vues et des procédures stockées soigneusement conçues pour ne lui exposer que le strict nécessaire.

Pourquoi la séparation physique ?

Le modèle classique consiste à donner à l'utilisateur applicatif un `GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.*`. C'est rapide à mettre en place, mais c'est une catastrophe sécuritaire :

- L'utilisateur a accès à **toutes** les colonnes de **toutes** les tables
- Une injection SQL dans l'application expose l'intégralité de la base
- Aucune traçabilité fine des accès aux données sensibles
- Impossible de masquer certaines colonnes (numéros de carte, emails, mots de passe hashés)

La séparation physique résout ces problèmes en interposant une **couche d'abstraction SQL** entre l'application et les données.

Étape 1 : Créer un schéma d'interface

```
CREATE DATABASE app_interface;
```

Ce schéma ne contiendra aucune table. Uniquement des vues et des procédures stockées. C'est la "surface d'attaque" visible par l'application.

Étape 2 : Créer des vues avec ALGORITHM=TEMPTABLE

La clé de la séparation physique réside dans le choix de l'algorithme de la vue :

```
CREATE
  ALGORITHM = TEMPTABLE
  DEFINER = 'admin_views'@'localhost'
  SQL SECURITY DEFINER
VIEW app_interface.v_customers AS
SELECT
    customer_id,
    first_name,
    last_name,
    city,
    country
FROM production.customers;
```

Trois éléments critiques ici :

- **ALGORITHM=TEMPTABLE** : MariaDB matérialise la vue dans une table temporaire. L'utilisateur ne peut pas "remonter" à la table sous-jacente via `SHOW CREATE VIEW` pour construire une requête directe.

- **DEFINER** : La vue s'exécute avec les droits du compte `admin_views`, pas ceux de l'utilisateur applicatif.
- **SQL SECURITY DEFINER** : Les vérifications de privilèges sont faites sur le DEFINER, pas sur l'INVOKER. L'utilisateur applicatif n'a besoin de droits que sur la vue, pas sur la table source.

Notez ce qui manque dans la vue : pas d'email, pas de numéro de téléphone, pas d'adresse complète. **Le masquage de données est intrinsèque à la conception de la vue.**

Étape 3 : Procédures stockées pour les écritures

Pour les opérations d'écriture, les procédures stockées offrent un contrôle encore plus fin :

```
DELIMITER //
CREATE PROCEDURE app_interface.sp_update_customer_city(
    IN p_customer_id INT,
    IN p_city VARCHAR(100)
)
SQL SECURITY DEFINER
BEGIN
    -- Validation métier
    IF p_city IS NULL OR LENGTH(TRIM(p_city)) = 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'City cannot be empty';
    END IF;

    UPDATE production.customers
    SET city = p_city,
        updated_at = NOW()
    WHERE customer_id = p_customer_id;

    -- Audit trail
    INSERT INTO production.audit_log(
        table_name, record_id, field_name,
        action, performed_by, performed_at
    )
    VALUES (
        'customers', p_customer_id, 'city',
        'UPDATE', CURRENT_USER(), NOW()
    );
END //
```

```
DELIMITER ;
```

L'utilisateur applicatif ne peut modifier que la ville. Pas le nom, pas l'email, pas le statut du compte. Et chaque modification est auditée automatiquement.

Étape 4 : Le compte administrateur verrouillé

Le compte DEFINER des vues et procédures ne doit jamais être utilisé pour se connecter :

```
CREATE USER 'admin_views'@'localhost'
    IDENTIFIED BY 'impossible_to_guess_random_string';

GRANT SELECT, INSERT, UPDATE ON production.* TO 'admin_views'@'localhost';

ALTER USER 'admin_views'@'localhost' ACCOUNT LOCK;
```

Un compte verrouillé (`ACCOUNT LOCK`) ne peut pas se connecter, mais ses privilèges restent actifs pour les vues et procédures en mode `SQL SECURITY DEFINER`. C'est le point crucial de l'architecture : **le compte qui a les droits ne peut pas se connecter, et le compte qui se connecte n'a pas les droits directs**.

Étape 5 : L'utilisateur applicatif minimal

```
CREATE USER 'app_user'@'10.0.%'
    IDENTIFIED BY 'strong_password_here';

GRANT SELECT ON app_interface.v_customers TO 'app_user'@'10.0.%';
GRANT EXECUTE ON PROCEDURE app_interface.sp_update_customer_city
    TO 'app_user'@'10.0.%';

-- Aucun GRANT sur production.*
```

L'utilisateur applicatif n'a accès à rien dans le schéma `production`. Même en cas d'injection SQL réussie, l'attaquant ne peut voir que les données exposées par les vues et ne peut exécuter que les procédures autorisées.

Masquage de données avancé

Les vues permettent aussi des techniques de masquage sophistiquées :

```
CREATE VIEW app_interface.v_customer_contacts AS
SELECT
    customer_id,
    CONCAT(LEFT(email, 3), '***@***.',
           SUBSTRING_INDEX(email, '.', -1)) AS masked_email,
    CONCAT('***-***-', RIGHT(phone, 4)) AS masked_phone
FROM production.customers;
```

Le support client peut identifier un client par les 4 derniers chiffres de son téléphone sans jamais voir le numéro complet.

Limitation du débit par requête

Une technique souvent négligée : utiliser les procédures stockées pour implémenter un rate limiting au niveau de la base :

```
CREATE PROCEDURE app_interface.sp_search_customers(
    IN p_search_term VARCHAR(100)
)
SQL SECURITY DEFINER
BEGIN
    DECLARE v_count INT;

    SELECT COUNT(*) INTO v_count
    FROM production.rate_limit
    WHERE user = CURRENT_USER()
      AND action = 'search'
      AND created_at > NOW() - INTERVAL 1 MINUTE;

    IF v_count > 10 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Rate limit exceeded: max 10 searches/minute';
    END IF;

    INSERT INTO production.rate_limit(user, action, created_at)
    VALUES (CURRENT_USER(), 'search', NOW());

    SELECT customer_id, first_name, last_name, city
```

```
    FROM production.customers
    WHERE last_name LIKE CONCAT(p_search_term, '%')
    LIMIT 50;
END;
```

Récapitulatif de l'architecture

Couche	Composant	Rôle
Application	`app_user`	Se connecte, exécute des vues/procédures
Interface	`app_interface` (schéma)	Expose uniquement les données nécessaires
Sécurité	`admin_views` (verrouillé)	Détient les droits, ne peut pas se connecter
Production	`production` (schéma)	Tables réelles, inaccessibles directement

Les limites

Cette approche n'est pas parfaite :

- **Performance** : `ALGORITHM=TEMPTABLE` crée une copie temporaire. Pour les grosses tables, cela peut être coûteux.
- **Complexité** : Chaque nouvelle fonctionnalité applicative nécessite potentiellement une nouvelle vue ou procédure.
- **Maintenance** : Les vues doivent évoluer avec le schéma des tables sous-jacentes.

Mais ces contraintes sont le prix de la sécurité. Et dans un contexte où les fuites de données coûtent en moyenne 4,5 millions de dollars par incident, c'est un investissement raisonnable.

Conclusion

La séparation physique via les vues `TEMPTABLE` et les procédures stockées `DEFINER` n'est pas une fonctionnalité obscure de MariaDB / MySQL. C'est une architecture de sécurité robuste, native, et souvent sous-exploitée.

Cinq étapes suffisent : un schéma d'interface, des vues avec le bon algorithme, des procédures pour les écritures, un compte DEFINER verrouillé, et un utilisateur applicatif minimal. Le résultat est une base de données où même une injection SQL réussie ne donne accès qu'à une fraction

contrôlée des données.

Cet article a été initialement publié sur Medium.