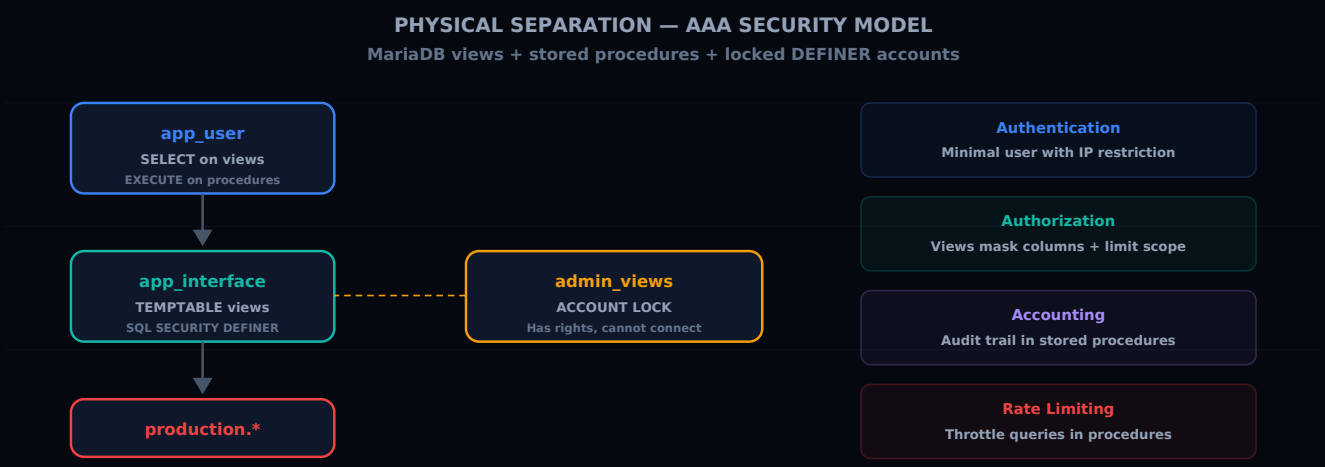
Разделимся физически

Sylvain ARBAUDIE · November 4, 2024

MARIADB SECURITY ACCESS-CONTROL VIEWS



Модель AAA применительно к базам данных

В информационной безопасности модель AAA (Authentication, Authorization, Accounting) — фундаментальная опора. Она встречается в RADIUS, TACACS+, файрволах, VPN... но редко применяется с должной строгостью к реляционным базам данных.

При этом MariaDB / MySQL предлагает нативные механизмы, позволяющие реализовать настоящую физическую изоляцию между конфиденциальными данными и пользователями приложения. Не нужно дополнительного middleware, не нужно дорогого прокси. Всё уже есть в движке.

Центральная идея проста: **прикладной пользователь никогда не должен иметь прямой доступ к таблицам с конфиденциальными данными**. Он должен взаимодействовать только с представлениями и хранимыми процедурами, тщательно разработанными для предоставления лишь строго необходимого.

Зачем физическое разделение?

Классическая модель — дать прикладному пользователю `GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.*`. Это быстро настраивается, но с точки зрения безопасности это

катастрофа:

- Пользователь имеет доступ ко **всем** столбцам **всех** таблиц
- SQL-инъекция в приложении обнажает всю базу данных
- Невозможна тонкая трассировка доступа к конфиденциальным данным
- Нельзя скрыть определённые столбцы (номера карт, email-адреса, хешированные пароли)

Физическое разделение решает эти проблемы, вставляя **слой абстракции SQL** между приложением и данными.

Этап 1: Создать схему интерфейса

```
CREATE DATABASE app_interface;
```

Эта схема не будет содержать таблиц. Только представления и хранимые процедуры. Это «поверхность атаки», видимая приложению.

Этап 2: Создать представления с ALGORITHM=TEMPTABLE

Ключ физического разделения — в выборе алгоритма представления:

```
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;
```

Три критических элемента:

- **ALGORITHM=TEMPTABLE**: MariaDB материализует представление во временную таблицу. Пользователь не может «подняться» к базовой таблице через `SHOW CREATE VIEW` для построения прямого запроса.
- **DEFINER**: Представление выполняется с правами учётной записи `admin_views`, а не прикладного пользователя.
- **SQL SECURITY DEFINER**: Проверки привилегий выполняются для DEFINER, а не для INVOKER. Прикладному пользователю нужны права только на представление, а не на исходную таблицу.

Обратите внимание на то, что отсутствует в представлении: ни email, ни номер телефона, ни полный адрес. **Маскирование данных является внутренним свойством дизайна представления.**

Этап 3: Хранимые процедуры для записи

Для операций записи хранимые процедуры предлагают ещё более тонкий контроль:

```
DELIMITER //
CREATE PROCEDURE app_interface.sp_update_customer_city(
    IN p_customer_id INT,
    IN p_city VARCHAR(100)
)
SQL SECURITY DEFINER
BEGIN
    -- Бизнес-валидация
    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;

    -- Журнал аудита
    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 ;
```

Прикладной пользователь может изменить только город. Не имя, не email, не статус учётной записи. И каждое изменение автоматически журналируется.

Этап 4: Заблокированная учётная запись администратора

Учётная запись DEFINER представлений и процедур никогда не должна использоваться для подключения:

```
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;
```

Заблокированная учётная запись (`ACCOUNT LOCK`) не может подключиться, но её привилегии остаются активными для представлений и процедур в режиме `SQL SECURITY DEFINER`. Это ключевой момент архитектуры: **учётная запись, обладающая правами, не может подключиться, а учётная запись, которая подключается, не имеет прямых прав**.

Этап 5: Минимальный прикладной пользователь

```
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.%';

-- Никаких GRANT на production.*
```

Прикладной пользователь не имеет доступа ни к чему в схеме `production`. Даже при успешной SQL-инъекции атакующий может видеть только данные, предоставленные представлениями, и выполнять только разрешённые процедуры.

Продвинутое маскирование данных

Представления также позволяют использовать изощрённые техники маскирования:

```
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;
```

Служба поддержки может идентифицировать клиента по последним 4 цифрам телефона, никогда не видя полный номер.

Ограничение частоты запросов

Часто упускаемая из виду техника: использование хранимых процедур для реализации rate limiting на уровне базы данных:

```
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;
```

Сводка архитектуры

Слой	Компонент	Роль
Приложение	`app_user`	Подключается, выполняет представления/процедуры
Интерфейс	`app_interface` (схема)	Предоставляет только необходимые данные
Безопасность	`admin_views` (заблокирован)	Обладает правами, не может подключиться
Продакшн	`production` (схема)	Реальные таблицы, недоступны напрямую

Ограничения

Этот подход не идеален:

- **Производительность**: `ALGORITHM=TEMPTABLE` создаёт временную копию. Для больших таблиц это может быть затратно.
- **Сложность**: Каждая новая функция приложения потенциально требует нового представления или процедуры.
- **Сопровождение**: Представления должны эволюционировать вместе со схемой базовых таблиц.

Но эти ограничения — цена безопасности. А в контексте, где утечки данных стоят в среднем 4,5 миллиона долларов за инцидент, это разумная инвестиция.

Заключение

Физическое разделение через представления `TEMPTABLE` и хранимые процедуры `DEFINER` — это не малоизвестная функция MariaDB / MySQL. Это надёжная, нативная и часто недоиспользуемая архитектура безопасности.

Пяти этапов достаточно: схема интерфейса, представления с правильным алгоритмом, процедуры для записи, заблокированная учётная запись DEFINER и минимальный прикладной пользователь. Результат — база данных, в которой даже успешная SQL-инъекция даёт доступ лишь к контролируемой доле данных.

Эта статья была первоначально опубликована на Medium.