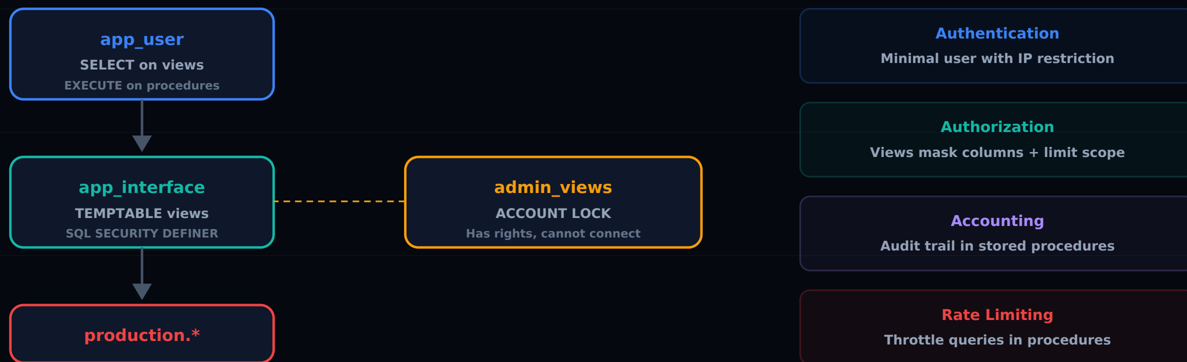
Oddzielmy się fizycznie

Sylvain ARBAUDIE · November 4, 2024

MARIADB SECURITY ACCESS-CONTROL VIEWS



Model AAA zastosowany do baz danych

W bezpieczeństwie informatycznym model AAA (Authentication, Authorization, Accounting) to fundamentalny filar. Spotykamy go w RADIUS, TACACS+, firewallach, VPN-ach... ale rzadko jest rygorystycznie stosowany do relacyjnych baz danych.

A jednak MariaDB / MySQL oferuje natywne mechanizmy pozwalające na implementację prawdziwej fizycznej separacji między danymi wrażliwymi a użytkownikami aplikacyjnymi. Nie potrzeba dodatkowego middleware'u, nie potrzeba kosztownego proxy. Wszystko jest już dostępne w silniku.

Centralna idea jest prosta: **użytkownik aplikacyjny nigdy nie powinien mieć bezpośredniego dostępu do tabel zawierających dane wrażliwe**. Powinien interagować wyłącznie z widokami i procedurami składowanymi starannie zaprojektowanymi, aby udostępniać mu tylko niezbędne minimum.

Dlaczego fizyczna separacja?

Klasyczny model polega na nadaniu użytkownikowi aplikacyjnemu `GRANT SELECT, INSERT, UPDATE, DELETE ON mydb.*`. To szybkie we wdrożeniu, ale katastrofalne pod względem bezpieczeństwa:

- Użytkownik ma dostęp do **wszystkich** kolumn **wszystkich** tabel
- Iniekcja SQL w aplikacji ujawnia całą bazę danych
- Brak szczegółowej śledzenia dostępu do danych wrażliwych
- Niemożność ukrycia pewnych kolumn (numery kart, emaile, hasła zahashowane)

Fizyczna separacja rozwiązuje te problemy, wprowadzając **warstwę abstrakcji SQL** między aplikacją a danymi.

Krok 1: Utworzenie schematu interfejsowego

```
CREATE DATABASE app_interface;
```

Ten schemat nie będzie zawierał żadnych tabel. Wyłącznie widoki i procedury składowane. To "powierzchnia ataku" widoczna dla aplikacji.

Krok 2: Tworzenie widoków z ALGORITHM=TEMPTABLE

Kluczem do fizycznej separacji jest wybór algorytmu widoku:

```
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;
```

Trzy krytyczne elementy:

- **ALGORITHM=TEMPTABLE**: MariaDB materializuje widok w tabeli tymczasowej. Użytkownik nie może "cofnąć się" do tabeli źródłowej przez `SHOW CREATE VIEW`, aby zbudować bezpośrednie zapytanie.

- **DEFINER**: Widok jest wykonywany z uprawnieniami konta `admin_views`, a nie użytkownika aplikacyjnego.
- **SQL SECURITY DEFINER**: Sprawdzanie uprawnień odbywa się na DEFINERZE, nie na INVOKERZE. Użytkownik aplikacyjny potrzebuje uprawnień tylko do widoku, nie do tabeli źródłowej.

Zwróćmy uwagę, czego brakuje w widoku: nie ma emaila, numeru telefonu, pełnego adresu. **Maskowanie danych jest wbudowane w projekt widoku.**

Krok 3: Procedury składowane do zapisów

Dla operacji zapisu procedury składowane oferują jeszcze dokładniejszą kontrolę:

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

    -- Ślad audytowy
    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 ;
```

Użytkownik aplikacyjny może modyfikować wyłącznie miasto. Nie nazwisko, nie email, nie status konta. A każda modyfikacja jest automatycznie audytowana.

Krok 4: Zablokowane konto administratora

Konto DEFINER widoków i procedur nigdy nie powinno być używane do logowania:

```
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;
```

Zablokowane konto (`ACCOUNT LOCK`) nie może się zalogować, ale jego uprawnienia pozostają aktywne dla widoków i procedur w trybie `SQL SECURITY DEFINER`. To kluczowy punkt architektury: **konto, które ma uprawnienia, nie może się zalogować, a konto, które się loguje, nie ma bezpośrednich uprawnień**.

Krok 5: Minimalny użytkownik aplikacyjny

```
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.%';

-- Żadne GRANT na production.*
```

Użytkownik aplikacyjny nie ma dostępu do niczego w schemacie `production`. Nawet w przypadku udanej iniekcji SQL atakujący może zobaczyć tylko dane udostępnione przez widoki i może wykonywać wyłącznie autoryzowane procedury.

Zaawansowane maskowanie danych

Widoki umożliwiają również zaawansowane techniki maskowania:

```
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;
```

Dział obsługi klienta może zidentyfikować klienta po 4 ostatnich cyfrach numeru telefonu, nigdy nie widząc pełnego numeru.

Ograniczanie przepustowości zapytań

Często pomijana technika: użycie procedur składowanych do implementacji rate limitingu na poziomie bazy danych:

```
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;
```

Podsumowanie architektury

Warstwa	Komponent	Rola
Aplikacja	`app_user`	Łączy się, wykonuje widoki/procedury
Interfejs	`app_interface` (schemat)	Udostępnia wyłącznie niezbędne dane
Bezpieczeństwo	`admin_views` (zablokowany)	Posiada uprawnienia, nie może się zalogować
Produkcja	`production` (schemat)	Rzeczywiste tabele, niedostępne bezpośrednio

Ograniczenia

To podejście nie jest idealne:

- **Wydajność**: `ALGORITHM=TEMPTABLE` tworzy tymczasową kopię. Dla dużych tabel może to być kosztowne.
- **Złożoność**: Każda nowa funkcjonalność aplikacyjna potencjalnie wymaga nowego widoku lub procedury.
- **Utrzymanie**: Widoki muszą ewoluować wraz ze schematem tabel źródłowych.

Ale te ograniczenia to cena bezpieczeństwa. A w kontekście, gdzie wycieki danych kosztują średnio 4,5 miliona dolarów za incydent, to rozsądna inwestycja.

Podsumowanie

Fizyczna separacja przez widoki `TEMPTABLE` i procedury składowane `DEFINER` to nie obscuryczna funkcja MariaDB / MySQL. To solidna, natywna architektura bezpieczeństwa, często niedostatecznie wykorzystywana.

Pięć kroków wystarczy: schemat interfejsowy, widoki z właściwym algorytmem, procedury do zapisów, zablokowane konto DEFINER i minimalny użytkownik aplikacyjny. Rezultatem jest baza danych, w której nawet udana iniekcja SQL daje dostęp jedynie do kontrolowanej frakcji danych.

Ten artykuł został pierwotnie opublikowany na Medium.