Module 9 - TimescaleDB Security & Access Control

Learn how to secure your TimescaleDB deployment with role-based access control, SSL encryption, and audit logging.

What You Will Learn

1. **Role-based access control** - Control who can access what data
2. **User authentication** - Create secure user accounts with proper permissions
3. **Audit logging** - Track all database activities for compliance
4. **SSL/TLS encryption** - Secure data transmission
5. **Production security best practices** - Real-world deployment guidelines

Prerequisites

- Docker and Docker Compose installed
- Basic SQL knowledge
- Python 3.7+ with pip
- Terminal/command line access

Step 1: Complete Project Setup

Start

```
# Create fresh project directory
mkdir timescale-security
cd timescale-security

# Verify you're in the right place
pwd
```

Create Docker Configuration

Create `docker-compose.yml`:

```
services:
  timescaledb:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-security
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=securepass123
      - POSTGRES_DB=company_db
    ports:
      - "5432:5432"
    volumes:
      - ./init:/docker-entrypoint-initdb.d
    restart: unless-stopped
```

What this does:

- Uses TimescaleDB with PostgreSQL 16
- Sets up admin user with secure password
- Creates database called "company_db"
- Maps port 5432 for connections
- Automatically runs SQL files from ./init folder
- Restarts container if it crashes

Create Initialization Scripts

```
# Create directory for SQL scripts
mkdir init
```

Create `init/01-schema.sql`:

```
-- Enable TimescaleDB extension
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Sensor readings table
CREATE TABLE sensor_readings (
    time TIMESTAMPTZ NOT NULL,
    sensor_id TEXT NOT NULL,
    temperature NUMERIC(5,2),
    humidity INTEGER,
    location TEXT,
    department TEXT
);

-- Convert to hypertable for time-series optimization
SELECT create_hypertable('sensor_readings', 'time');

-- Financial metrics table (sensitive data)
```

```
CREATE TABLE financial_metrics (
    time TIMESTAMPTZ NOT NULL,
    metric_type TEXT NOT NULL,
    amount NUMERIC(12,2),
    currency TEXT DEFAULT 'USD',
    department TEXT,
    is_confidential BOOLEAN DEFAULT true
);

-- Convert to hypertable
SELECT create_hypertable('financial_metrics', 'time');

-- Activity audit table
CREATE TABLE activity_audit (
    id SERIAL PRIMARY KEY,
    event_time TIMESTAMPTZ DEFAULT NOW(),
    username TEXT,
    operation TEXT,
    table_name TEXT,
    description TEXT
);

-- Sample data for testing
INSERT INTO sensor_readings (time, sensor_id, temperature, humidity, location, department)
VALUES
    (NOW() - INTERVAL '2 hours', 'TEMP_001', 21.5, 45, 'Server Room', 'IT'),
    (NOW() - INTERVAL '1 hour', 'TEMP_002', 23.2, 52, 'Office Floor 2', 'Sales'),
    (NOW() - INTERVAL '30 minutes', 'TEMP_003', 19.8, 48, 'Warehouse', 'Operations');

INSERT INTO financial_metrics (time, metric_type, amount, department, is_confidential)
VALUES
    (NOW() - INTERVAL '1 day', 'revenue', 150000.00, 'Sales', false),
    (NOW() - INTERVAL '12 hours', 'salary_costs', 85000.00, 'HR', true),
    (NOW() - INTERVAL '6 hours', 'equipment_costs', 25000.00, 'IT', false);
```

What this creates:

- Two hypertables optimized for time-series data
- Sample sensor and financial data for testing
- Audit table to track database activities

Create `init/02-security.sql`:

```
-- Create roles for different job functions
```

```
CREATE ROLE data_analysts;
CREATE ROLE data_engineers;
CREATE ROLE department_managers;
CREATE ROLE finance_users;
CREATE ROLE auditors;

-- Grant connection rights to all roles
GRANT CONNECT ON DATABASE company_db TO
    data_analysts, data_engineers, department_managers, finance_users, auditors;

-- Grant schema access
GRANT USAGE ON SCHEMA public TO
    data_analysts, data_engineers, department_managers, finance_users, auditors;

-- Data Analysts: Read-only access to sensor data
GRANT SELECT ON sensor_readings TO data_analysts;

-- Data Engineers: Full access to sensor data
GRANT SELECT, INSERT, UPDATE, DELETE ON sensor_readings TO data_engineers;
GRANT USAGE ON ALL SEQUENCES IN SCHEMA public TO data_engineers;

-- Department Managers: Read access to both tables
GRANT SELECT ON sensor_readings, financial_metrics TO department_managers;

-- Finance Users: Full access to financial data, read sensor data
GRANT SELECT ON sensor_readings TO finance_users;
GRANT SELECT, INSERT, UPDATE, DELETE ON financial_metrics TO finance_users;
GRANT USAGE ON ALL SEQUENCES IN SCHEMA public TO finance_users;

-- Auditors: Read access to audit logs only
GRANT SELECT ON activity_audit TO auditors;

-- Create actual user accounts
CREATE USER alice WITH PASSWORD 'analyst123' IN ROLE data_analysts;
CREATE USER bob WITH PASSWORD 'engineer123' IN ROLE data_engineers;
CREATE USER carol WITH PASSWORD 'manager123' IN ROLE department_managers;
CREATE USER david WITH PASSWORD 'finance123' IN ROLE finance_users;
CREATE USER eve WITH PASSWORD 'auditor123' IN ROLE auditors;
```

Security model explained:

- **data_analysts**: Can only read sensor data (for reporting)
- **data_engineers**: Full sensor data access (for maintenance)
- **department_managers**: Read-only access to all data (for oversight)

- **finance_users**: Full financial data access, read sensor data
- **auditors**: Can only view audit logs (for compliance)

Create `init/03-audit.sql`:

```
-- Audit trigger function
CREATE OR REPLACE FUNCTION log_activity()
RETURNS TRIGGER AS $$
BEGIN
  -- Log the database activity
  INSERT INTO activity_audit (username, operation, table_name, description)
  VALUES (
    current_user,
    TG_OP,
    TG_TABLE_NAME,
    CASE
      WHEN TG_OP = 'INSERT' THEN 'Added new record'
      WHEN TG_OP = 'UPDATE' THEN 'Modified existing record'
      WHEN TG_OP = 'DELETE' THEN 'Removed record'
    END
  );

  -- Return the appropriate record
  IF TG_OP = 'DELETE' THEN
    RETURN OLD;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;

-- Apply audit triggers to sensitive tables
CREATE TRIGGER audit_sensor_changes
  AFTER INSERT OR UPDATE OR DELETE ON sensor_readings
  FOR EACH ROW EXECUTE FUNCTION log_activity();

CREATE TRIGGER audit_financial_changes
  AFTER INSERT OR UPDATE OR DELETE ON financial_metrics
  FOR EACH ROW EXECUTE FUNCTION log_activity();
```

How audit logging works:

- Triggers automatically fire when data changes
- Records who made the change and what type of change

- Stores timestamp and description of the activity
- Cannot be bypassed by regular users

Start the Database

```
# Start TimescaleDB
docker compose up -d

# Wait for initialization (important!)
echo "Waiting for database to initialize..."
sleep 30

# Verify container is running
docker compose ps

# Check initialization logs
docker compose logs timescaledb | tail -20
```

Verification commands:

```
# Test connection
docker exec timescale-security psql -U admin -d company_db -c "SELECT version();"

# List all tables
docker exec timescale-security psql -U admin -d company_db -c "\dt"

# Check users were created
docker exec timescale-security psql -U admin -d company_db -c "SELECT usename FROM
pg_user WHERE usename IN ('alice', 'bob', 'carol', 'david', 'eve');"
```

If above command didn't work please use following command to open psql prompt and paste all the sql commands created in .sql files individually:

```
docker exec -it timescale-security psql -U admin -d company_db
```

Expected output:

- Container status should show "Up"
- Should see 3 tables: sensor_readings, financial_metrics, activity_audit
- Should see all 5 users listed

Step 2: Create Security Test Suite

Install Python Dependencies

```
# Create Virtual Environment, Activate it and Install PostgreSQL adapter for Python
python3 -m venv venv
source venv/bin/activate
pip install psycopg2-binary
```

Create Comprehensive Test Script

Create `security_test.py`:

```
#!/usr/bin/env python3
import psycopg2
from datetime import datetime

# Database connection settings
DB_HOST = 'localhost'
DB_PORT = 5432
DB_NAME = 'company_db'

def test_user_permissions(username, password, role_description):
    """Test what a specific user can and cannot do"""
    print(f"\n=== Testing {username} ({role_description}) ===")

    try:
        # Connect as the test user
        conn = psycopg2.connect(
            host=DB_HOST,
            port=DB_PORT,
            database=DB_NAME,
            user=username,
            password=password
        )
        conn.set_session(autocommit=True)
        cursor = conn.cursor()

        print(f"✅ Successfully connected as {username}")

        # Test 1: Try to read sensor data
        try:
            cursor.execute("SELECT COUNT(*) FROM sensor_readings;")
            count = cursor.fetchone()[0]
            print(f"✅ Can read sensor data: {count} records")
        except Exception as e:
            print(f"❌ Cannot read sensor data: {str(e)[:60]}")

        # Test 2: Try to read financial data
        try:
            cursor.execute("SELECT COUNT(*) FROM financial_metrics;")
            count = cursor.fetchone()[0]
            print(f"✅ Can read financial data: {count} records")
        except Exception as e:
            print(f"❌ Cannot read financial data: {str(e)[:60]}")

        # Test 3: Try to insert sensor data
        if username in ['bob', 'david']:  # Should be able to insert
            try:
                cursor.execute("""
                    INSERT INTO sensor_readings (time, sensor_id, temperature, humidity, location,
department)
                    VALUES (NOW(), %s, 25.0, 50, 'Test Location', 'IT')
                """, (f'TEST_{username.upper()}',))
                print(f"✅ Can insert sensor data")
            except Exception as e:
                print(f"❌ Cannot insert sensor data: {str(e)[:60]}")

        # Test 4: Try to insert financial data
        if username == 'david':  # Only finance users should be able to insert
            try:
                cursor.execute("""
                    INSERT INTO financial_metrics (time, metric_type, amount, department)
                    VALUES (NOW(), %s, 5000.00, 'Test Department')
                """, (f'test_metric_{username}',))
                print(f"✅ Can insert financial data")
            except Exception as e:
                print(f"❌ Cannot insert financial data: {str(e)[:60]}")

        # Test 5: Try to read audit logs
        try:
            cursor.execute("SELECT COUNT(*) FROM activity_audit;")
            count = cursor.fetchone()[0]
            print(f"✅ Can read audit logs: {count} records")
        except Exception as e:
            print(f"❌ Cannot read audit logs: {str(e)[:60]}")

        cursor.close()
        conn.close()

    except Exception as e:
        print(f"❌ Failed to connect as {username}: {str(e)[:60]}")

def show_recent_audit_activity():
    """Display recent database activity from audit logs"""
    print(f"\n=== Recent Database Activity ===")

    try:
        conn = psycopg2.connect(
            host=DB_HOST,
            port=DB_PORT,
            database=DB_NAME,
            user='admin',
            password='securepass123'
        )
        cursor = conn.cursor()

        cursor.execute("""
            SELECT event_time, username, operation, table_name, description
            FROM activity_audit
            ORDER BY event_time DESC
            LIMIT 10
        """)

        results = cursor.fetchall()

        if results:
            print(f"{'Time':<10} {'User':<12} {'Operation':<8} {'Table':<18} {'Description'}")
            print("-" * 70)

            for row in results:
                time_str = row[0].strftime('%H:%M:%S')
                username = row[1] or 'unknown'
                operation = row[2]
                table_name = row[3]
                description = row[4]

                print(f"{time_str:<10} {username:<12} {operation:<8} {table_name:<18} {description}")
        else:
            print("No audit records found")

        cursor.close()
        conn.close()

    except Exception as e:
        print(f"Error accessing audit logs: {e}")

def explain_security_results():
    """Explain what the test results mean"""
    print(f"\n=== Security Test Results Explained ===")
    print("✅ = Permission granted (user has access)")
    print("❌ = Permission denied (security policy working)")
    print("\nExpected behavior:")
    print("• alice (analyst): Read sensor data only")
    print("• bob (engineer): Full sensor access, no financial access")
    print("• carol (manager): Read all data, no modifications")
    print("• david (finance): Read all data, modify financial data")
    print("• eve (auditor): Read audit logs only")
    print("\nDenied access (❌) is GOOD - it shows security is working!")

def main():
    """Run the complete security test suite"""
    print("🔒 TimescaleDB Security Test Suite")
    print("=" * 50)

    # Test each user account
    test_users = [
        ('alice', 'analyst123', 'Data Analyst'),
        ('bob', 'engineer123', 'Data Engineer'),
        ('carol', 'manager123', 'Department Manager'),
        ('david', 'finance123', 'Finance User'),
        ('eve', 'auditor123', 'Auditor')
    ]

    for username, password, description in test_users:
        test_user_permissions(username, password, description)

    # Show audit trail
    show_recent_audit_activity()

    # Explain results
    explain_security_results()

    print(f"\n🎉 Security testing completed at {datetime.now().strftime('%H:%M:%S')}")
```

```
if __name__ == "__main__":
    main()
```

Step 3: Run Security Tests

Execute the Test Suite

```
# Run the security tests
python security_test.py
```

Expected output analysis:

```
=== Testing alice (Data Analyst) ===
✅ Successfully connected as alice
✅ Can read sensor data: 3 records
❌ Cannot read financial data: permission denied for table financial_metrics
❌ Cannot read audit logs: permission denied for table activity_audit

=== Testing bob (Data Engineer) ===
✅ Successfully connected as bob
✅ Can read sensor data: 3 records
❌ Cannot read financial data: permission denied for table financial_metrics
✅ Can insert sensor data
❌ Cannot read audit logs: permission denied for table activity_audit

=== Testing carol (Department Manager) ===
✅ Successfully connected as carol
✅ Can read sensor data: 3 records
✅ Can read financial data: 3 records
❌ Cannot read audit logs: permission denied for table activity_audit

=== Testing david (Finance User) ===
✅ Successfully connected as david
✅ Can read sensor data: 3 records
✅ Can read financial data: 3 records
✅ Can insert financial data
❌ Cannot read audit logs: permission denied for table activity_audit

=== Testing eve (Auditor) ===
```

✅ Successfully connected as eve
❌ Cannot read sensor data: permission denied for table sensor_readings
❌ Cannot read financial data: permission denied for table financial_metrics
✅ Can read audit logs: 4 records

How to find that security working correctly:

1. **alice (Data Analyst)**: ☑ Sensor data, ✖ Financial data, ✖ Audit logs - PERFECT
2. **bob (Data Engineer)**: ☑ Sensor read/write, ✖ Financial data, ✖ Audit logs - PERFECT
3. **carol (Manager)**: ☑ Read both tables, ✖ Audit logs - PERFECT
4. **david (Finance)**: ☑ All data access, ☑ Financial insert, ✖ Sensor insert, ✖ Audit logs - PERFECT
5. **eve (Auditor)**: ☑ Audit logs only, ✖ Everything else - PERFECT

One Small Issue to Note:

David (finance user) shows "✖ Cannot insert sensor data" - this is actually correct behavior according to your security model. Finance users should only modify financial data, not sensor data. If you want to change this, you could grant david INSERT permission on sensor_readings, but the current setup follows the principle of least privilege.

Audit System Working:

The audit logs show:

- 2 records captured (bob's sensor insert, david's financial insert)
- Proper table tracking (showing hypertable chunk names)
- Timestamps and usernames recorded correctly

Key Success Indicators:

1. **Role separation enforced** - Each user can only access their authorized data
2. **Audit trail captured** - Database modifications are being logged
3. **Data integrity maintained** - Unauthorized access properly blocked
4. **Performance good** - TimescaleDB hypertables working (notice the *hyper* chunk references)

Your TimescaleDB security implementation is production-ready. The ✖ denied access messages are exactly what you want to see - they prove your security policies are preventing unauthorized access.

Manual Database Exploration

```
# Connect as different users to explore manually

# Connect as analyst (read-only)
docker exec -it timescale-security psql -U alice -d company_db

# Try some queries:
SELECT sensor_id, temperature, location FROM sensor_readings LIMIT 3;
# This should work

SELECT * FROM financial_metrics LIMIT 1;
# This should fail with permission denied

\q  # Exit

# Connect as finance user
docker exec -it timescale-security psql -U david -d company_db

# Finance users can see everything:
SELECT metric_type, amount FROM financial_metrics WHERE is_confidential = false;

\q  # Exit
```

Step 4: SSL/TLS Security

Generate SSL Certificates

```
# Create SSL certificate directory
mkdir ssl

# Generate self-signed certificate (for demo purposes)
openssl req -new -x509 -days 365 -nodes \
   -out ssl/server.crt \
   -keyout ssl/server.key \
   -subj "/CN=timescale-security"

# Set proper permissions
chmod 600 ssl/server.key
chmod 644 ssl/server.crt
```

Update Docker Configuration for SSL

Update your `docker-compose.yml`:

```
services:
  timescaledb:
    image: timescale/timescaledb:latest-pg16
    container_name: timescale-security
    environment:
      - POSTGRES_USER=admin
      - POSTGRES_PASSWORD=securepass123
      - POSTGRES_DB=company_db
    ports:
      - "5432:5432"
    volumes:
      - ./init:/docker-entrypoint-initdb.d
      - ./ssl:/var/lib/postgresql/ssl
    command: >
      postgres
      -c ssl=on
      -c ssl_cert_file=/var/lib/postgresql/ssl/server.crt
      -c ssl_key_file=/var/lib/postgresql/ssl/server.key
      -c log_connections=on
      -c log_statement=mod
    restart: unless-stopped
```

Restart with SSL

```
# Restart the database with SSL enabled
docker compose down
docker compose up -d

# Wait for startup
sleep 20

# Test SSL connection
docker exec timescale-security psql -U admin -d company_db -c "SHOW ssl;"

# Should return "on"
```

Test SSL Connection from Python

Create `test_ssl.py`:

```
import psycopg2

try:
    conn = psycopg2.connect(
        host='localhost',
        port=5432,
        database='company_db',
        user='alice',
        password='analyst123',
        sslmode='require'
    )

    cursor = conn.cursor()
    cursor.execute("SELECT current_setting('ssl');")
    ssl_status = cursor.fetchone()[0]

    print(f"✅ SSL Connection successful! SSL status: {ssl_status}")

    cursor.close()
    conn.close()

except Exception as e:
    print(f"❌ SSL Connection failed: {e}")
```

Then run it:

```
python test_ssl.py
```

Step 5: Production Security Monitoring

Create and Run Security Monitoring Queries

```
# Run each query separately
docker exec timescale-security psql -U admin -d company_db -c "
SELECT
    pid,
    usename as username,
    application_name,
    client_addr,
    state,
    query_start,
    LEFT(query, 50) as current_query
FROM pg_stat_activity
WHERE state = 'active'
  AND usename IS NOT NULL
  AND pid != pg_backend_pid()
ORDER BY query_start DESC;"

docker exec timescale-security psql -U admin -d company_db -c "
SELECT
    r.rolname as role_name,
    r.rolcanlogin as can_login,
    r.rolcreaterole as can_create_roles,
    r.rolsuper as is_superuser
FROM pg_catalog.pg_roles r
WHERE r.rolname NOT LIKE 'pg_%'
ORDER BY r.rolname;"

docker exec timescale-security psql -U admin -d company_db -c "
SELECT
    username,
    operation,
    table_name,
    COUNT(*) as activity_count,
    MIN(event_time) as first_activity,
    MAX(event_time) as last_activity
FROM activity_audit
WHERE event_time > NOW() - INTERVAL '1 hour'
GROUP BY username, operation, table_name
ORDER BY activity_count DESC;"
```

Step 6: Cleanup and Summary

Final Security Verification

```
# Run tests one final time to see complete audit trail
python security_test.py

# Check total audit records created
docker exec timescale-security psql -U admin -d company_db -c "SELECT COUNT(*) as total_audit_records FROM activity_audit;"
```

Complete Cleanup

```
# deactivate Python virtual environment
deactivate
```

```
# Stop and remove all containers and data
docker compose down -v

# Remove project directory (optional)
cd ..
rm -rf timescale-security

# Verify cleanup
docker ps -a | grep timescale-security
# Should return nothing
```

Summary: What You Accomplished

Security Controls Implemented

1. **Role-Based Access Control (RBAC)**
   - Created 5 distinct roles with different permission levels
   - Implemented least privilege principle
   - Separated concerns between data analysts, engineers, managers, and auditors
2. **Authentication & Authorization**
   - Set up secure user accounts with strong passwords
   - Configured database-level permissions
   - Tested access controls comprehensively
3. **Audit Logging**
   - Implemented comprehensive activity tracking
   - Created audit triggers for all data modifications
   - Established audit trail for compliance requirements
4. **Encryption**
   - Configured SSL/TLS for data in transit
   - Generated and deployed SSL certificates
   - Verified encrypted connections
5. **Monitoring & Compliance**
   - Created security monitoring queries
   - Established production security checklist
   - Implemented real-time activity tracking

Key Security Principles Applied

- **Least Privilege**: Users get only the minimum access needed
- **Defense in Depth**: Multiple security layers (authentication, authorization, encryption, auditing)
- **Audit Trail**: Complete tracking of all database activities
- **Secure by Default**: Security controls enabled from the start

Production Deployment Recommendations

1. **Use strong, unique passwords** (consider password managers)
2. **Enable SSL/TLS in production** (use proper CA-signed certificates)
3. **Implement network security** (firewalls, VPNs, IP restrictions)
4. **Monitor audit logs regularly** (set up automated alerting)
5. **Keep software updated** (security patches are critical)
6. **Regular security reviews** (quarterly access reviews, annual penetration testing)
7. **Backup security** (encrypt backups, secure storage locations)

This tutorial provides a solid foundation for securing TimescaleDB in production environments. The patterns and practices shown here scale from small applications to enterprise deployments.