**ASSESSMENT 5 – SECURITY WHIZ**

For this QA security assessment, I used MySQL to demonstrate how common database-level controls can be implemented and validated. However, if SoftMouse.NET runs on PostgreSQL in production, I would recommend using pgcrypto, Row-Level Security, and pgAudit to meet higher standards for research data protection and traceability.

Colony Module – Database Security Controls Proposal

Technology Assumed: MySQL (with optional PostgreSQL considerations)

**Control 1: Data Encryption at Rest**

***What is the control?***

Enable encryption of sensitive colony data (e.g., genotypes, cage history, mating logs) while stored in the database using AES encryption or Transparent Data Encryption (TDE).

***Why do we need this control?***

If the database files are accessed by unauthorized users (e.g., due to a server compromise), encryption ensures the raw data remains unreadable.

***How to test the control as a QA?***

1. Ensure encryption is enabled via:

SHOW VARIABLES LIKE 'have\_encryption';

1. Attempt to read encrypted table data from disk — it should not be human-readable.
2. Verify that authorized queries correctly decrypt data through the application UI.

**Control 2: Role-Based Access Control (RBAC)**

***What is the control?***

Restrict database access to specific tables and actions using user roles (e.g., ReadOnly, LabTech, Admin), controlled via GRANT/REVOKE permissions.

***Why do we need this control?***

This prevents data leaks or accidental changes. For example, a LabTech should not be able to modify pedigree records or access billing information.

***How to test the control as a QA?***

1. Log in with users assigned to each role.
2. Attempt unauthorized operations (e.g., UPDATE as a read-only user).
3. Ensure access is denied with appropriate error messages.

**Control 3: Audit Logging for Sensitive Tables**

***What is the control?***

Track all queries and operations (especially SELECT, INSERT, DELETE) performed on sensitive tables like mice, pedigree, and users.

***Why do we need this control?***

Enables traceability and incident response in case of a breach, accidental deletion, or unauthorized access.

***How to test the control as a QA?***

1. Enable general query logging.
2. Execute sample queries on sensitive tables.
3. Review the logs to confirm that query, user, and timestamp are recorded.

**Why I Preferred MySQL for This Assessment**

1. Simple to Set Up and Use for Local Testing - I already installed and configured MySQL, which allowed me to quickly demonstrate security test cases.
2. Sufficient for Functional Demos - MySQL offers enough features like encryption functions and access control to simulate real-world scenarios without additional tools.
3. Good Community Support - Easier to find help and examples for automation integration and basic auditing.

**What If PostgreSQL Is Used Instead?**

If the backend database is PostgreSQL, the same security principles still apply, but with some different tools and techniques. Below is a comparison and additional suggestions:

**MySQL → PostgreSQL Equivalents**

|  |  |  |
| --- | --- | --- |
| **Security Control Area** | **MySQL** | **PostgreSQL** |
| **Encryption** | AES encryption, TDE (Enterprise) | pgcrypto for column-level encryption |
| **Access Control** | GRANT / REVOKE permissions | Row-Level Security (RLS) for fine-grained access |
| **Audit Logging** | General logs, logging plugins | pgAudit extension for advanced auditing |

**Additional PostgreSQL-Specific Security Recommendations**

* **Row-Level Security (RLS):**  
  Use RLS to restrict access to specific rows in a table based on user roles or identities.  
  *Example:* Only show animals owned by the lab the user belongs to.
* **pgAudit Plugin:**  
  Install and configure pgAudit to log all read/write operations on sensitive tables such as animals, matings, or genotypes.
* **SSL/TLS Enforcement:**  
  Enforce encrypted connections between application and database to prevent data leakage during transmission.

**Summary**

While MySQL and PostgreSQL use different tools, both support strong database-level security when properly configured. PostgreSQL offers more flexibility in fine-grained access control (via RLS) and structured auditing (via pgAudit), making it an excellent choice for regulated or multi-tenant environments.