Comprehensive Security Analysis: Evaluating Basic Authentication in MariaDB and MongoDB

Patrick Lunney
Melbourne, Australia
patrick-lunney@outlook.com
October 2024

Abstract—**This project presents a comparative analysis of the security features in MariaDB and MongoDB, with a primary focus on their authentication mechanisms. Implemented within a simulated healthcare data management scenario, the study evaluates the ease of setup, password management, and user access control of MariaDB's ed25519 and MongoDB's SCRAM authentication mechanisms. Through practical testing I assess the strengths and limitations of each system, offering recommendations for enhancing security practices in both systems.**

Keywords—Database Security, MariaDB, MongoDB, Authentication, Access Control, ed25519, SCRAM, SQL, NoSQL

I. Introduction

With the rise of digital security threats in 2024, database security has become a critical aspect of database management, particularly in industries handling sensitive information such as healthcare. Health service providers were the top sector to report data breaches, accounting for 19% of all notifications, with breaches often involving malicious attacks or human error [1]. This highlights the importance of securing sensitive healthcare data, as breaches can lead to severe legal or financial consequences.

Database security encompasses the tools and measures that ensure the confidentiality, integrity, and availability of data – principles known as the CIA Triad, which form the foundation of modern security practices [2], [3]. As organizations handle increasing amounts of sensitive data, implementing robust security measures is vital to prevent breaches and maintaining customer trust.

This report compares the security features of MariaDB (a relational SQL database) and MongoDB (a NoSQL database) within the context of managing healthcare data. The comparison covers five critical aspects of database security: encryption, authentication, authorization, auditing, and injection vulnerability protection [4], [5]. My practical testing emphasizes authentication, using a simulated healthcare data management system to demonstrate the importance of securing sensitive information.

II. Overview of MariaDB and MongoDB

A. *MariaDB: A Relational SQL Database.*

1) Database Type and Origins.

MariaDB, an open-source relational database management system (RDBMS) was created as a fork of MySQL in 2009 to remain free after Oracle's acquisition of MySQL. Licensed under GNU GPLv2, it is widely used in major Linux distributions and offers SQL-based querying with transactional support [6], [7].

2) Current Use Cases.

MariaDB is popular for applications requiring structured data management, high data integrity, and transactions. Its MySQL compatibility makes it a preferred choice for organizations seeking easy migration or upgrades without major code changes [6], [7].

B. *MongoDB: A NoSQL Database.*

1) Database Type and Origins.

Developed in 2007, MongoDB is an open-source, document-oriented NoSQL database that stores data in JSON-like documents with flexible schemas. Its scalable design supports modern applications with high performance and rapid data processing [5], [8].

2) Current Use Cases.

MongoDB is ideal for applications handling large, diverse datasets and high scalability, such as IoT, mobile apps, content management, and real-time analytics. Its flexible data model efficiently manages unstructured data [5], [8].

III. Security Features of MariaDB and MongoDB

A. *Authentication:*

1) The Importance of Authentication

Authentication ensures that only authorized users can access the database, preventing unauthorized access to sensitive information. Testing authentication confirms that the database properly identifies and verifies users, which is essential for database security [9].

2) Authentication in MariaDB

MariaDB manages authentication through plugins, with legacy options like mysql_native_password using the weaker SHA1 hash function to store passwords [10]. It offers various methods, including password-based, TLS, and OS-based authentication via plugins (e.g., named_pipe) [11], [12]. While these add security, vulnerabilities can arise, highlighting the need for secure configurations and regular updates. [10].

3) Authentication in MongoDB:

MongoDB supports several authentication mechanisms, including SCRAM (default), x.509, LDAP, and Kerberos authentication to verify user and application identities [5], [8], [13], [14]. These options allow MongoDB to integrate with existing systems, ensuring only authorized users can access the database and protecting sensitive information [9].

B. Encryption:

1) The Importance of Encryption

Testing encryption ensures data security during transit and at rest, preventing unauthorized access or tampering. Encryption encodes sensitive data, allowing only those with the correct decryption key to access or interpret it. [9]. Testing verifies that strong methods are in place to protect data at all stages.

2) Encryption in MariaDB:

MariaDB offers data-at-rest encryption, particularly in its enterprise version, but it is not enabled by default and requires additional configuration [10]. For data-in-motion, MariaDB supports TLS/SSL and x.509 certificate-based encryption to secure communication between clients and servers [8], [10].

3) Encryption in MongoDB:

MongoDB encrypts data both in transit and at rest. It uses TLS/SSL for data-in-transit encryption between clients and servers [5], [8], [13], [14]. Data-at-rest encryption is provided via the WiredTiger storage engine, available in MongoDB Enterprise or MongoDB Atlas [5], [8], [13], [14]. MongoDB also supports client-side field-level encryption, allowing users to encrypt specific fields before transmission [8], [15].

C. Authorisation:

1) The Importance of Authorisation

Testing authorisation ensures users have access only to permitted resources and operations. This enforces security policies and the principle of least privilege, reducing the risk of insider threats or accidental data breaches [9].

2) Authorization in MariaDB:

MariaDB enforces authorization by default, with users granted privileges only when explicitly assigned [10]. This role-based access control ensures users must be authenticated and authorized before accessing or modifying data, supporting the principle of least privilege [10].

3) Authorization in MongoDB:

MongoDB uses Role-Based Access Control (RBAC) to manage permissions, assigning roles that determine access to resources and operations. Predefined roles such as dbAdmin, dbOwner, and clusterAdmin, can be customized for organizational needs [5], [8], [13], [14]. RBAC ensures users have only the necessary permissions, reinforcing the least privilege principle [14].

D. Auditing and Logging

1) The Importance of Auditing and Logging.

Testing ensures security events are properly recorded for compliance, forensic analysis, and detecting suspicious activities. Effective auditing and logging help monitor database activities, identify incidents promptly, and meet regulatory requirements [9].

2) Auditing and Logging in MariaDB:

MariaDB includes an audit plugin that provides policy-based auditing to enhance security and meet compliance requirements [10]. Audit logs record actions like authentication attempts and data modifications, essential for monitoring and investigating security incidents [10].

3) Auditing and Logging in MongoDB:

MongoDB Enterprise offers advanced auditing, logging actions such as administrative actions (DDL), authentication, authorization, and data operations (DML) [5], [8], [14], [15]. Audit trails can be filtered, and logs exported to multiple formats for analysis and compliance [15], [14].

E. Injection Vulnerability Protection

1) The Importance of Injection Vulnerability Protection.

Testing for injection vulnerabilities is crucial, as injection attacks can allow unauthorized access or command execution. Secure input validation and query handling protect the database from SQL or NoSQL injection attacks, preventing data theft, corruption, or loss [9], [16].

2) Injection vulnerability protection in MariaDB:

MariaDB is susceptible to SQL injection if user inputs aren't properly validated. Attackers can exploit these vulnerabilities to run unauthorized SQL commands [10]. Mitigation includes using prepared statements, input validation, and parameterized queries [10].

3) Injection vulnerability protection in MongoDB:

MongoDB is vulnerable to NoSQL injection attacks if user inputs aren't properly validated. Attackers can manipulate queries with malicious code, potentially leading to unauthorized access or data manipulation [5], [10], [16]. Proper input validation and parameterized queries mitigate these risks [5], [16].

IV. Testing Methodology

This report tests the password-based authentication mechanisms of MariaDB and MongoDB in a simulated healthcare database environment, focusing on securing sensitive patient data. The authentication methods tested are ed25519 for MariaDB and SCRAM for MongoDB, both assessed at their default configurations.

My evaluation will focus on two main criteria:

1) Ease of setup

Assessing time, configuration complexity, and challenges in setting up ed25519 and SCRAM.

2) Security robustness

Testing vulnerabilities, including bypass attempts, password expiration, and account locking.

The objective is to determine the effectiveness of each system in protecting sensitive data, particularly in the context of healthcare, where robust security measures are critical.

V. Database Setup and Initial Configuration

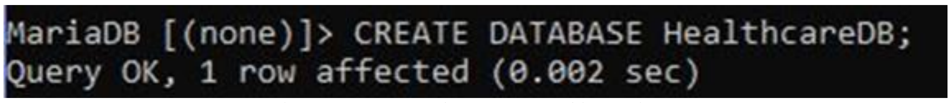
To test the authentication mechanisms of MariaDB and MongoDB, I created a simulated healthcare database environment. This setup reflects real-world scenarios, focusing on securing sensitive patient data. The database design is intentionally simple, highlighting essential elements needed to test authentication without unnecessary complexity.

A. Setting Up MariaDB

I installed MariaDB Community Server version 11-5-2-GA from the official website, using default settings and setting a secure master password for authentication.

B. Creating the Healthcare Database

I initiated this by creating a new database named HealthcareDB [17]. This database serves as the central repository for the simulated healthcare data.



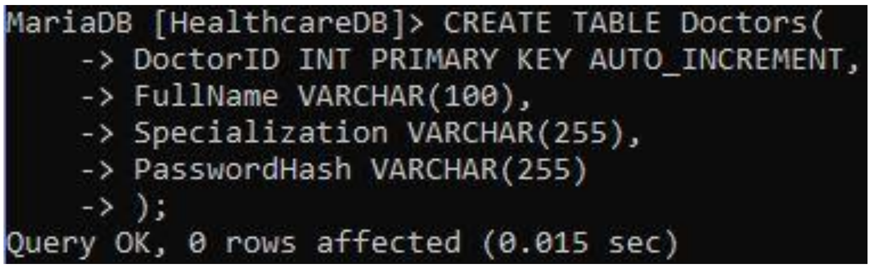
```
MariaDB [(none)]> CREATE DATABASE HealthcareDB;
Query OK, 1 row affected (0.002 sec)
```

Fig. 1. Create Database Command in MariaDB

To emulate a realistic healthcare environment, I created two fundamental tables: Doctors and Patients. The schema was deliberately kept straightforward to focus on the authentication mechanisms

1) Doctors Table:

The Doctors table stores essential information about medical professionals along with a placeholder for password hashes[18]. While not used in my authentication tests (handled via CREATE USER), the column reflects how sensitive data like password hashes might be stored in a real-world database for additional security

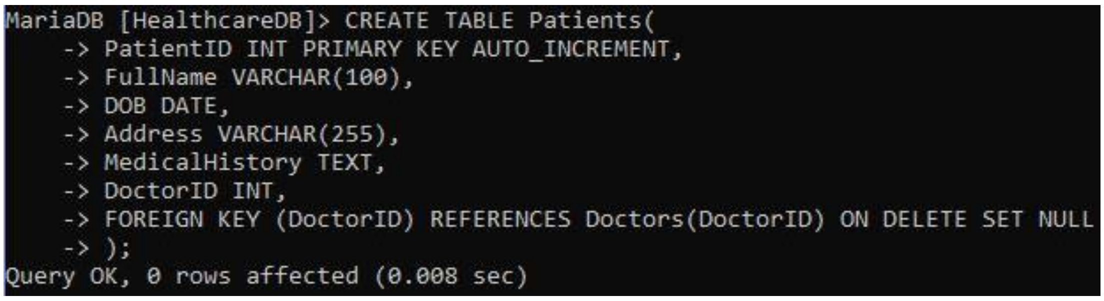


```
MariaDB [HealthcareDB]> CREATE TABLE Doctors(
    -> DoctorID INT PRIMARY KEY AUTO_INCREMENT,
    -> FullName VARCHAR(100),
    -> Specialization VARCHAR(255),
    -> PasswordHash VARCHAR(255)
    -> );
Query OK, 0 rows affected (0.015 sec)
```

Fig. 2. Created Doctors Table with Schema in MariaDB

2) Patients Table:

The Patients table contains patient information, linking each patient to a doctor via a foreign key constraint. This relationship ensures referential integrity within the database [18].

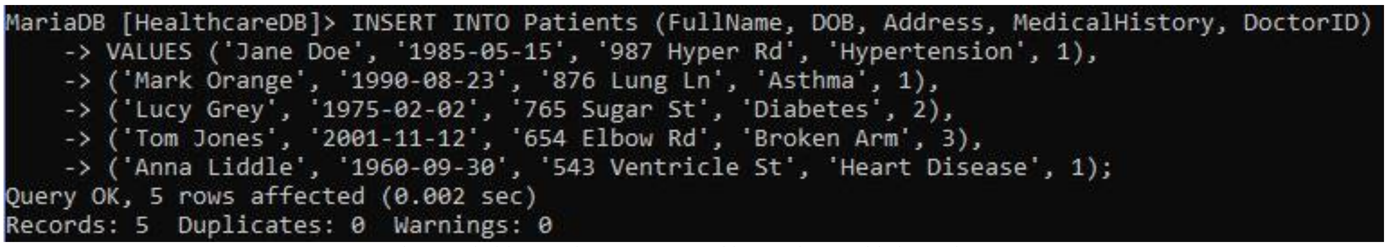


```
MariaDB [HealthcareDB]> CREATE TABLE Patients(
    -> PatientID INT PRIMARY KEY AUTO_INCREMENT,
    -> FullName VARCHAR(100),
    -> DOB DATE,
    -> Address VARCHAR(255),
    -> MedicalHistory TEXT,
    -> DoctorID INT,
    -> FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID) ON DELETE SET NULL
    -> );
Query OK, 0 rows affected (0.008 sec)
```

Fig. 3. Created Patients Table with Schema in MariaDB

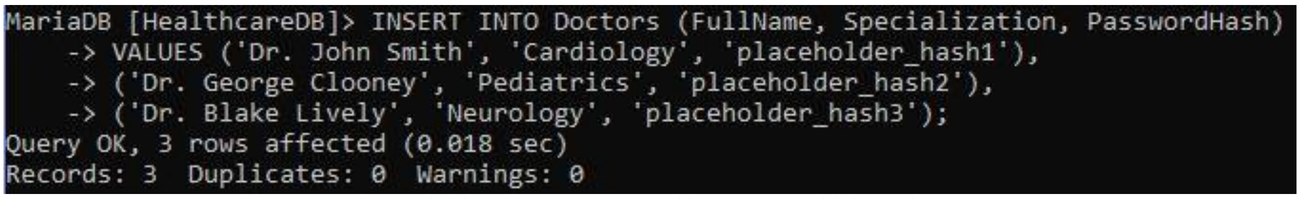
3) Populating the Tables with Sample Data

To facilitate meaningful testing, I inserted sample records into both tables using fictitious data. For the Doctors table, placeholder password hashes were used, acknowledging that these would be replaced if required once authentication mechanisms were implemented.



```
MariaDB [HealthcareDB]> INSERT INTO Patients (FullName, DOB, Address, MedicalHistory, DoctorID)
    -> VALUES ('Jane Doe', '1985-05-15', '987 Hyper Rd', 'Hypertension', 1),
    -> ('Mark Orange', '1990-08-23', '876 Lung Ln', 'Asthma', 1),
    -> ('Lucy Grey', '1975-02-02', '765 Sugar St', 'Diabetes', 2),
    -> ('Tom Jones', '2001-11-12', '654 Elbow Rd', 'Broken Arm', 3),
    -> ('Anna Liddle', '1960-09-30', '543 Ventricle St', 'Heart Disease', 1);
Query OK, 5 rows affected (0.002 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Fig. 4. Fictitious Patient Data Inserted into Patients Table in MariaDB



```
MariaDB [HealthcareDB]> INSERT INTO Doctors (FullName, Specialization, PasswordHash)
    -> VALUES ('Dr. John Smith', 'Cardiology', 'placeholder_hash1'),
    -> ('Dr. George Clooney', 'Pediatrics', 'placeholder_hash2'),
    -> ('Dr. Blake Lively', 'Neurology', 'placeholder_hash3');
Query OK, 3 rows affected (0.018 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Fig. 5. Fictitious Doctor Data Inserted into Doctors Table in MariaDB

C. Setting up MongoDB

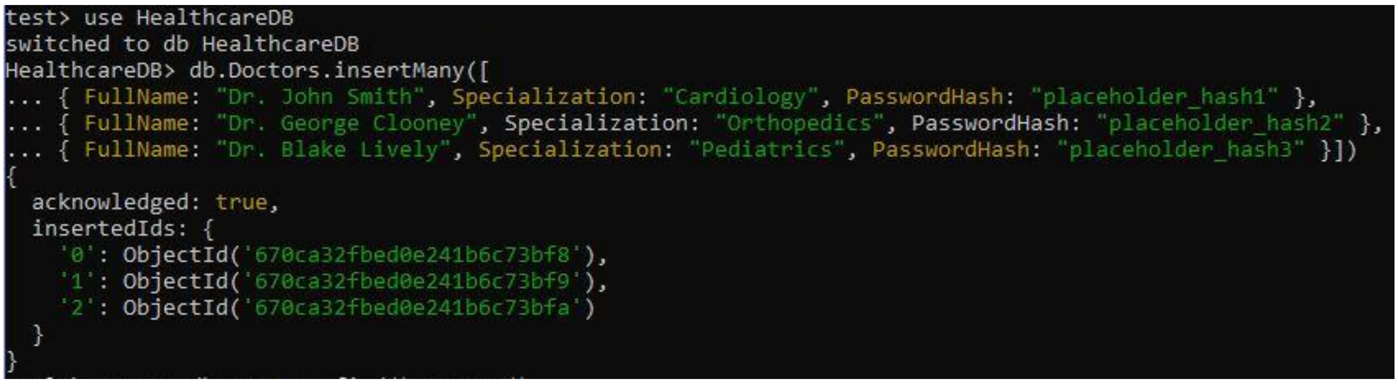
I installed MongoDB version 8.0.0 from the official website, configured it as a service for automatic startup, and installed MongoDB Compass (GUI) and Mongosh (shell). The system PATH was updated for easier terminal access [19].

D. Creating the Healthcare Database

In MongoDB databases are created when data is first inserted. I created HealthCareDB by inserting sample records into the Doctors collection (equivalent to a table in SQL-based databases), adapting its structure from the SQL schema used in MariaDB, with the help of the MongoDB documentation [20].

1) Doctors Collection:

I began by inserting records for the same three doctors into the Doctors collection [21]:

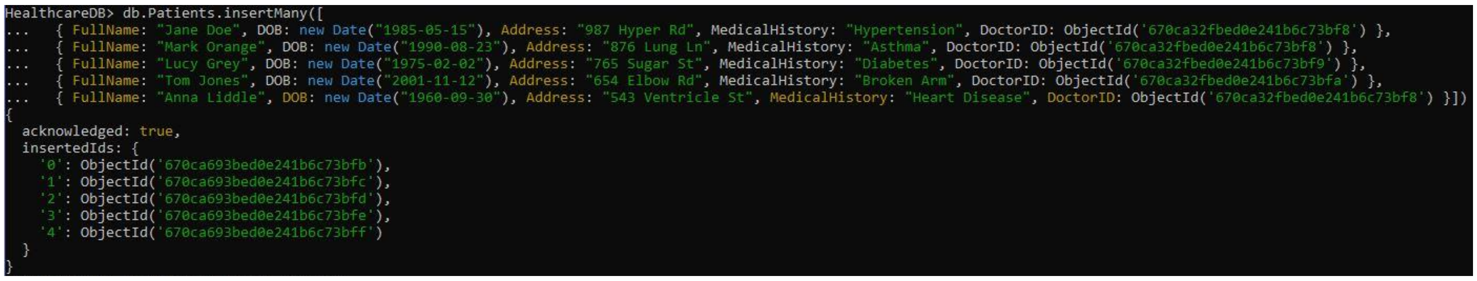


```
test> use HealthcareDB
switched to db HealthcareDB
HealthcareDB> db.Doctors.insertMany([
... { FullName: "Dr. John Smith", Specialization: "Cardiology", PasswordHash: "placeholder_hash1" },
... { FullName: "Dr. George Clooney", Specialization: "Orthopedics", PasswordHash: "placeholder_hash2" },
... { FullName: "Dr. Blake Lively", Specialization: "Pediatrics", PasswordHash: "placeholder_hash3" }])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('670ca32fbed0e241b6c73bf8'),
    '1': ObjectId('670ca32fbed0e241b6c73bf9'),
    '2': ObjectId('670ca32fbed0e241b6c73bfa')
  }
}
```

Fig. 6. HealthcareDB setup and Fictitious Doctor Data in MongoDB

2) Patients Collection:

Following the same process, I created the Patients collection, ensuring each patient was linked to a doctor using the DoctorID (represented by an ObjectId in MongoDB) [22].



```
HealthcareDB> db.Patients.insertMany([
...   { FullName: "Jane Doe", DOB: new Date("1985-05-15"), Address: "987 Hyper Rd", MedicalHistory: "Hypertension", DoctorID: ObjectId('670ca32fbed0e241b6c73bf8') },
...   { FullName: "Mark Orange", DOB: new Date("1990-08-23"), Address: "876 Lung Ln", MedicalHistory: "Asthma", DoctorID: ObjectId('670ca32fbed0e241b6c73bf8') },
...   { FullName: "Lucy Grey", DOB: new Date("1975-02-02"), Address: "765 Sugar St", MedicalHistory: "Diabetes", DoctorID: ObjectId('670ca32fbed0e241b6c73bf9') },
...   { FullName: "Tom Jones", DOB: new Date("2001-11-12"), Address: "654 Elbow Rd", MedicalHistory: "Broken Arm", DoctorID: ObjectId('670ca32fbed0e241b6c73bfa') },
...   { FullName: "Anna Liddle", DOB: new Date("1960-09-30"), Address: "543 Ventricle St", MedicalHistory: "Heart Disease", DoctorID: ObjectId('670ca32fbed0e241b6c73bf8') }])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('670ca693bed0e241b6c73bfb'),
    '1': ObjectId('670ca693bed0e241b6c73bfc'),
    '2': ObjectId('670ca693bed0e241b6c73bfd'),
    '3': ObjectId('670ca693bed0e241b6c73bfe'),
    '4': ObjectId('670ca693bed0e241b6c73bff')
  }
}
```

Fig. 7. Fictitious Patient Data inserted into the Patients Collection in MongoDB

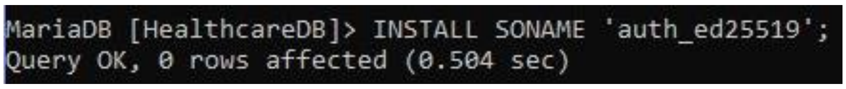
The collections created – Doctors and Patients – serve as the foundation for testing the authentication mechanisms in MongoDB, mirroring the data structure previously set up in MariaDB [23].

VI. Authentication Testing in MariaDB

This section explores the implementation of ed25519 authentication in MariaDB along with other MariaDB default variables and assesses their effectiveness through targeted tests. The aim is to evaluate how well this default authentication method secures user access, focusing on ease of setup, password handling and resistance to common vulnerabilities.

A. Setting up and Implementing ed25519 Authentication:

The ed25519 plugin, which provides secure password authentication using the elliptic curve digital signature algorithm (ECDSA), was manually installed [12], [24].

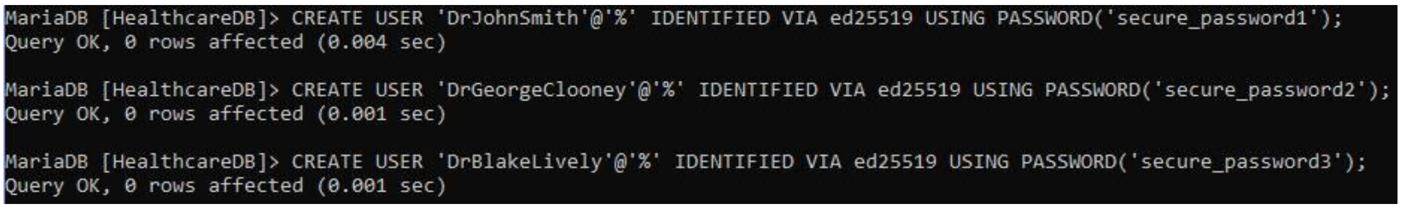


```
MariaDB [HealthcareDB]> INSTALL SONAME 'auth_ed25519';
Query OK, 0 rows affected (0.504 sec)
```

Fig. 8. Command to install the ed25519 plugin

B. Creating Users with ed25519:

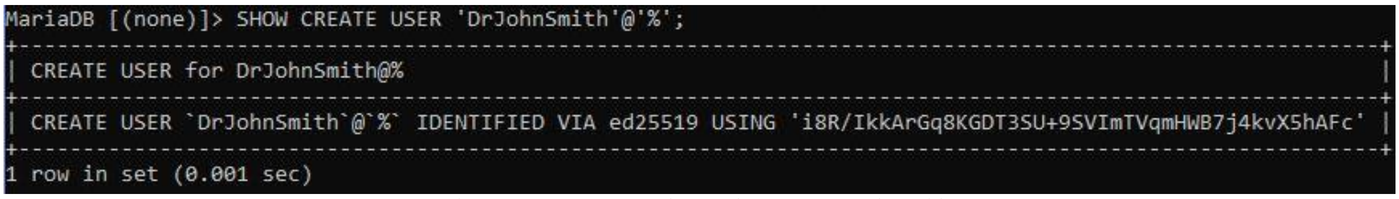
Creating users with ed25519 authentication was straightforward. Each user was associated with a password using the following commands [11]:



```
MariaDB [HealthcareDB]> CREATE USER 'DrJohnSmith'@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secure_password1');
Query OK, 0 rows affected (0.004 sec)

MariaDB [HealthcareDB]> CREATE USER 'DrGeorgeClooney'@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secure_password2');
Query OK, 0 rows affected (0.001 sec)

MariaDB [HealthcareDB]> CREATE USER 'DrBlakeLively'@'%' IDENTIFIED VIA ed25519 USING PASSWORD('secure_password3');
Query OK, 0 rows affected (0.001 sec)
```

Fig. 9. Doctor User Accounts created for Authentication Testing

This created three user accounts, each identified using the ed25519 algorithm. Confirmation of this could be obtained using the 'SHOW CREATE USER' statement, which displayed the associated ed25519 password hashes.



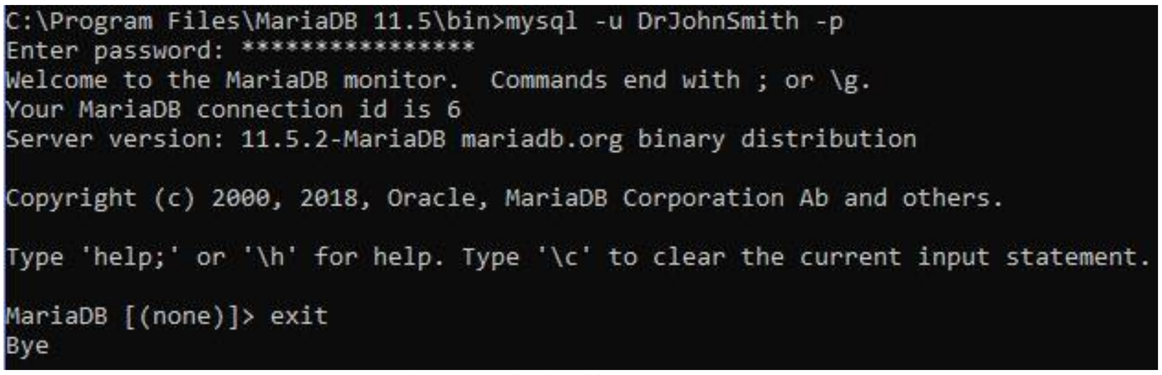
```
MariaDB [(none)]> SHOW CREATE USER 'DrJohnSmith'@'%';
+-----------------------------------------------------------------------------------------------------+
| CREATE USER for DrJohnSmith@%                                                                        |
+-----------------------------------------------------------------------------------------------------+
| CREATE USER `DrJohnSmith`@`%` IDENTIFIED VIA ed25519 USING 'i8R/IkkArGq8KGDT3SU+9SVImTVqmHWB7j4kvX5hAFc' |
+-----------------------------------------------------------------------------------------------------+
1 row in set (0.001 sec)
```

Fig. 10. SHOW Command example on User 'DrJohnSmith'

C. Authentication Tests:

1) Test 1: Successful Login.

I tested the login process for each user with their respective password. This was done by attempting to log in via the MariaDB client using the command(s):



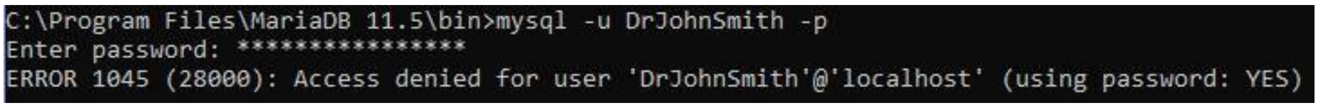
```
C:\Program Files\MariaDB 11.5\bin>mysql -u DrJohnSmith -p
Enter password: ****************
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 6
Server version: 11.5.2-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> exit
Bye
```

Fig. 11. Logging in with a Doctor Account

All users were able to log in successfully, confirming that the authentication mechanism was functioning as intended.

2) Test 2: Invalid Password Handling

Next, I attempted to log in using invalid passwords. In all cases, MariaDB rejected the login attempts with the appropriate error message, showing that the ed25519 mechanism properly handles incorrect credentials:

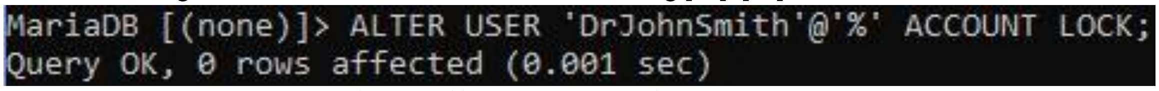


```
C:\Program Files\MariaDB 11.5\bin>mysql -u DrJohnSmith -p
Enter password: ****************
ERROR 1045 (28000): Access denied for user 'DrJohnSmith'@'localhost' (using password: YES)
```

Fig. 12. Error message when logging in with Incorrect Password

3) Test 3: Account Locking.

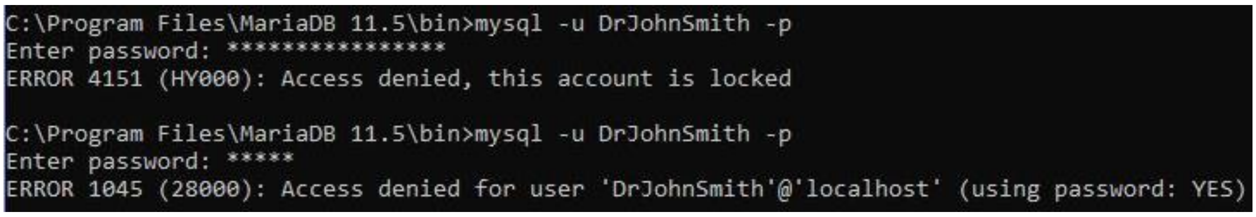
To test account locking, I locked the user account DrJohnSmith using [25], [26]:



```
MariaDB [(none)]> ALTER USER 'DrJohnSmith'@'%' ACCOUNT LOCK;
Query OK, 0 rows affected (0.001 sec)
```

Fig. 13. Locking Doctor Account using ALTER USER

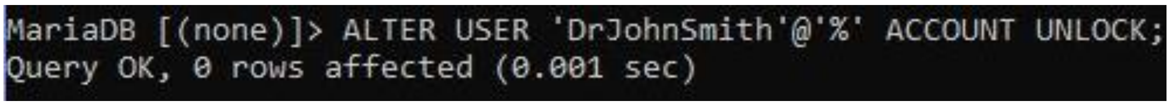
Subsequent login attempts were rejected with an appropriate error message. When the correct password was used, an account-locked message was presented, when an incorrect password was used, an incorrect password message was presented:



```
C:\Program Files\MariaDB 11.5\bin>mysql -u DrJohnSmith -p
Enter password: ****************
ERROR 4151 (HY000): Access denied, this account is locked

C:\Program Files\MariaDB 11.5\bin>mysql -u DrJohnSmith -p
Enter password: *****
ERROR 1045 (28000): Access denied for user 'DrJohnSmith'@'localhost' (using password: YES)
```

Fig. 14. Attempted Login to Locked Account using Correct and Incorrect Passwords

The account was then unlocked, and was able to be logged back into using:



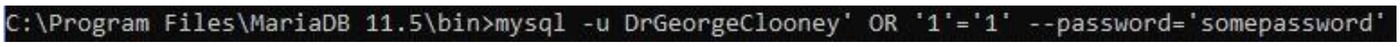
```
MariaDB [(none)]> ALTER USER 'DrJohnSmith'@'%' ACCOUNT UNLOCK;
Query OK, 0 rows affected (0.001 sec)
```

Fig. 15. Unlocking Doctor Account using ALTER USER

Returning functionality back to DrJohnSmith's account.

4) Test 4: SQL Injection Test.

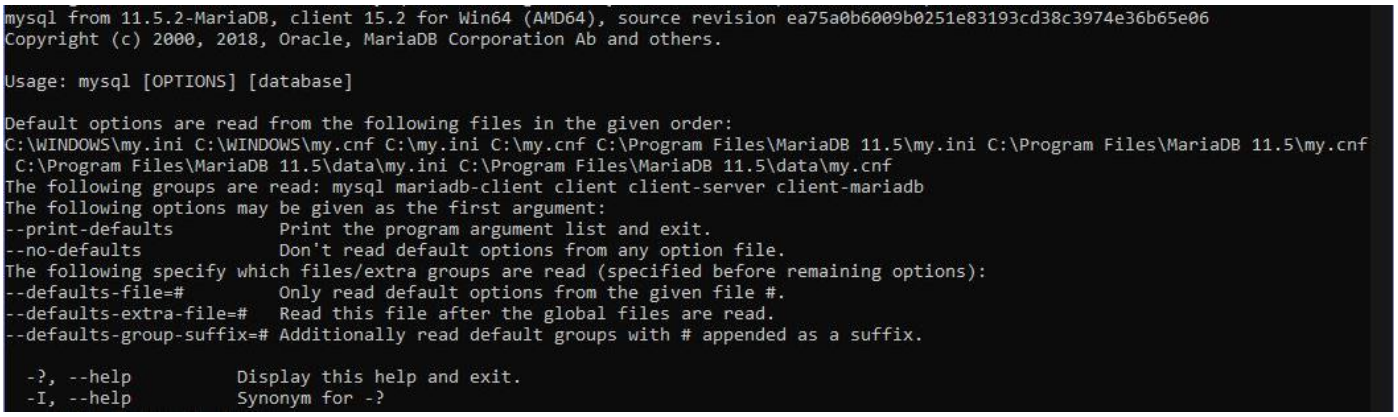
To test for SQL injection vulnerabilities, I attempted a typical injection at login [27]:



```
C:\Program Files\MariaDB 11.5\bin>mysql -u DrGeorgeClooney' OR '1'='1' --password='somepassword'
```

Fig. 16. Basic SQL Injection Code

This resulted in a help message from MariaDB, indicating that the injection attempt was unsuccessful. The input was treated as a malformed command rather than compromising the authentication mechanism, confirming that the system was resistant to this type of attack.

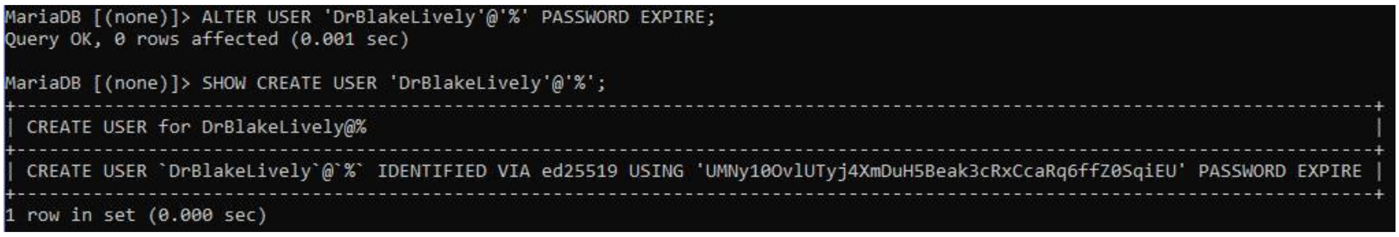


```
mysql from 11.5.2-MariaDB, client 15.2 for Win64 (AMD64), source revision ea75a0b6009b0251e83193cd38c3974e36b65e06
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Usage: mysql [OPTIONS] [database]

Default options are read from the following files in the given order:
C:\WINDOWS\my.ini C:\WINDOWS\my.cnf C:\my.ini C:\my.cnf C:\Program Files\MariaDB 11.5\my.ini C:\Program Files\MariaDB 11.5\my.cnf
 C:\Program Files\MariaDB 11.5\data\my.ini C:\Program Files\MariaDB 11.5\data\my.cnf
The following groups are read: mysql mariadb-client client client-server client-mariadb
The following options may be given as the first argument:
--print-defaults          Print the program argument list and exit.
--no-defaults             Don't read default options from any option file.
The following specify which files/extra groups are read (specified before remaining options):
--defaults-file=#         Only read default options from the given file #.
--defaults-extra-file=#   Read this file after the global files are read.
--defaults-group-suffix=# Additionally read default groups with # appended as a suffix.

  -?, --help          Display this help and exit.
  -I, --help          Synonym for -?
```

Fig. 17. MariaDB Responded to the SQL Injection with this Response

5) Test 5: Password Expiry.

To test password expiry, I began by manually expiring the password for the DrBlakeLively user [26], [28]:

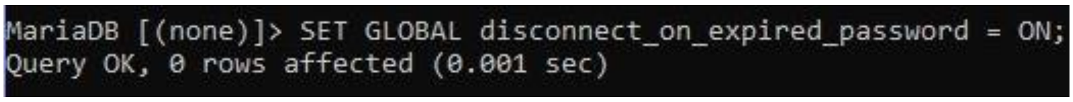


```
MariaDB [(none)]> ALTER USER 'DrBlakeLively'@'%' PASSWORD EXPIRE;
Query OK, 0 rows affected (0.001 sec)

MariaDB [(none)]> SHOW CREATE USER 'DrBlakeLively'@'%';
+-----------------------------------------------------------------------------------------------------------------------------+
| CREATE USER for DrBlakeLively@%                                                                                             |
+-----------------------------------------------------------------------------------------------------------------------------+
| CREATE USER `DrBlakeLively`@`%` IDENTIFIED VIA ed25519 USING 'UMNy10OvlUTyj4XmDuH5Beak3cRxCcaRq6ffZ0SqiEU' PASSWORD EXPIRE |
+-----------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.000 sec)
```

Fig. 18. Expiring Doctor User's Password using ALTER USER

Upon initial login, no error or prompt appeared, allowing the user to access the database. Even after logging out and back in again, the user was still able to access the database without being required to change the password. This behaviour represents a potential security gap, as the principle of least privilege would expect a user with an expired password to have no access until it is changed.

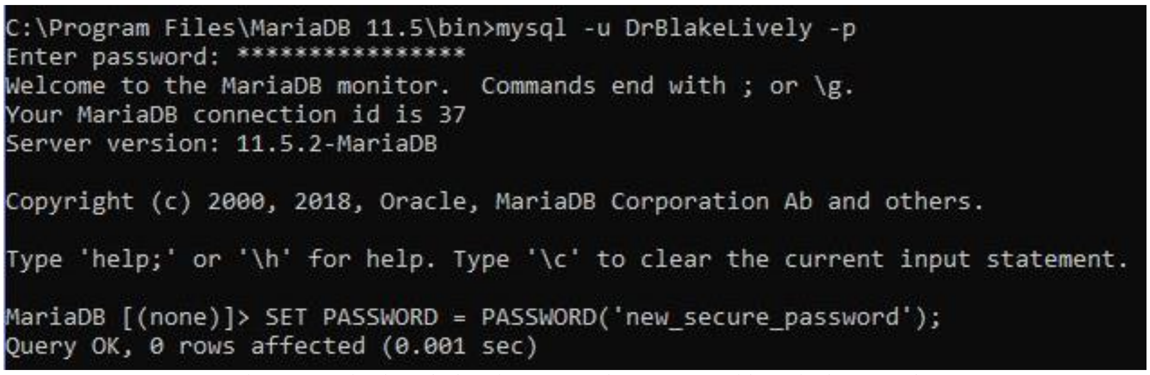
With further investigation, I discovered that MariaDB includes a global setting, disconnect_on_expired_password, which enforces password expiration policies more strictly. By default, this setting was turned off, which explained the lack of any feedback after the password had expired. I enabled this setting using the following command [28], [29]:



```
MariaDB [(none)]> SET GLOBAL disconnect_on_expired_password = ON;
Query OK, 0 rows affected (0.001 sec)
```

Fig. 19. Code for Changing Expired Password Global Variable

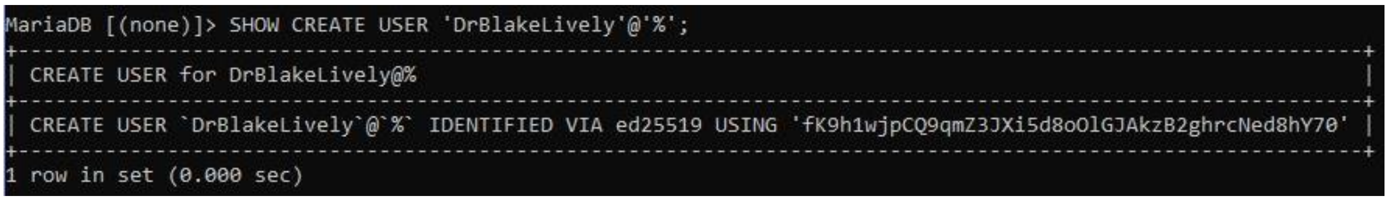
After enabling the global setting, I logged in to DrBlakeLively's account with the regular password. Unexpectedly, there was still no visual indication or message informing the user that their password had expired. Despite the lack of feedback, I was able to change the password by running [30]:



```
C:\Program Files\MariaDB 11.5\bin>mysql -u DrBlakeLively -p
Enter password: ****************
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 37
Server version: 11.5.2-MariaDB

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> SET PASSWORD = PASSWORD('new_secure_password');
Query OK, 0 rows affected (0.001 sec)
```

Fig. 20. Logging in and Changing Password on Account with Expired Password

I verified that the password change had taken effect by checking the hash via the root user account, and after logging out and back in everything was working as expected [28].

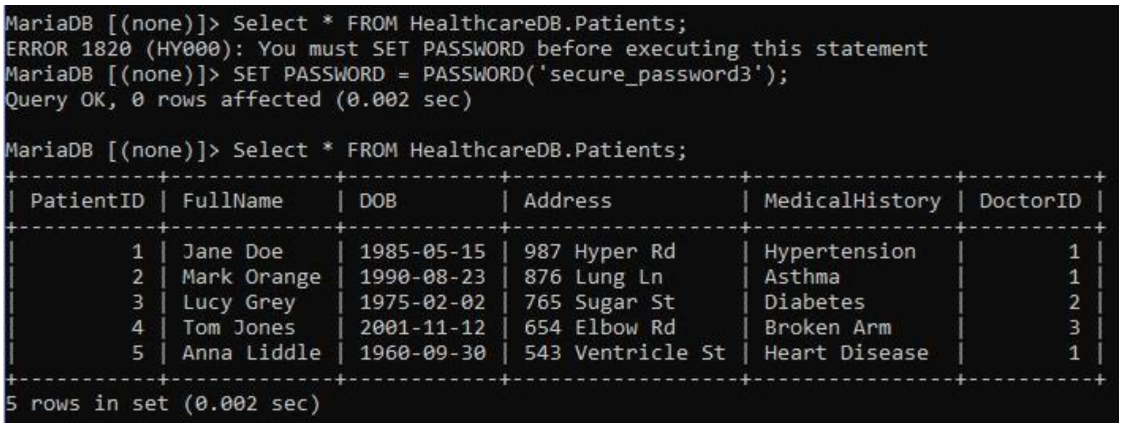


```
MariaDB [(none)]> SHOW CREATE USER 'DrBlakeLively'@'%';
+-------------------------------------------------------------------------------------------------+
| CREATE USER for DrBlakeLively@%                                                                 |
+-------------------------------------------------------------------------------------------------+
| CREATE USER `DrBlakeLively`@`%` IDENTIFIED VIA ed25519 USING 'fK9h1wjpCQ9qmZ3JXi5d8oOlGJAkzB2ghrcNed8hY70' |
+-------------------------------------------------------------------------------------------------+
1 row in set (0.000 sec)
```

Fig. 21. Checking Doctor Password after using SET PASSWORD

Following this, I tested to see if I could just change the password without having an expired password, and I was able to, using the same commands [30]. After further research I discovered that MariaDB, by default, allows users to change their password regardless of the global enforced setting [30], which raised doubt that the expired password setting was working as intended. As I continued looking into the issue, I uncovered that with the current configuration, once an expired password has been implemented, the account in question should be placed into a 'sandbox' mode where it is unable to perform actions on the database [28], [29].

With this knowledge, I added slight privileges to DrBlakeLively's account using the GRANT command [31] to allow SEARCH queries on the database, and then expired the password. I then tested to see if the sandbox mode (expired password) state was in effect:



```
MariaDB [(none)]> Select * FROM HealthcareDB.Patients;
ERROR 1820 (HY000): You must SET PASSWORD before executing this statement
MariaDB [(none)]> SET PASSWORD = PASSWORD('secure_password3');
Query OK, 0 rows affected (0.002 sec)

MariaDB [(none)]> Select * FROM HealthcareDB.Patients;
+-----------+-------------+------------+------------------+----------------+----------+
| PatientID | FullName    | DOB        | Address          | MedicalHistory | DoctorID |
+-----------+-------------+------------+------------------+----------------+----------+
|         1 | Jane Doe    | 1985-05-15 | 987 Hyper Rd     | Hypertension   |        1 |
|         2 | Mark Orange | 1990-08-23 | 876 Lung Ln      | Asthma         |        1 |
|         3 | Lucy Grey   | 1975-02-02 | 765 Sugar St     | Diabetes       |        2 |
|         4 | Tom Jones   | 2001-11-12 | 654 Elbow Rd     | Broken Arm     |        3 |
|         5 | Anna Liddle | 1960-09-30 | 543 Ventricle St | Heart Disease  |        1 |
+-----------+-------------+------------+------------------+----------------+----------+
5 rows in set (0.002 sec)
```

Fig. 22. User Attempts to use a SEARCH Query while Password is Expired

As observed in the above figure, the account was indeed placed into the sandbox mode as it was not able to perform a SEARCH on the database. I also was able to change the password, and then that re-enabled functionality to the account – verifying the password had been changed and was no longer expired. A critical finding was that the system permitted the new password to be identical to the previous one, highlighting a lack of password history enforcement that will be discussed further.

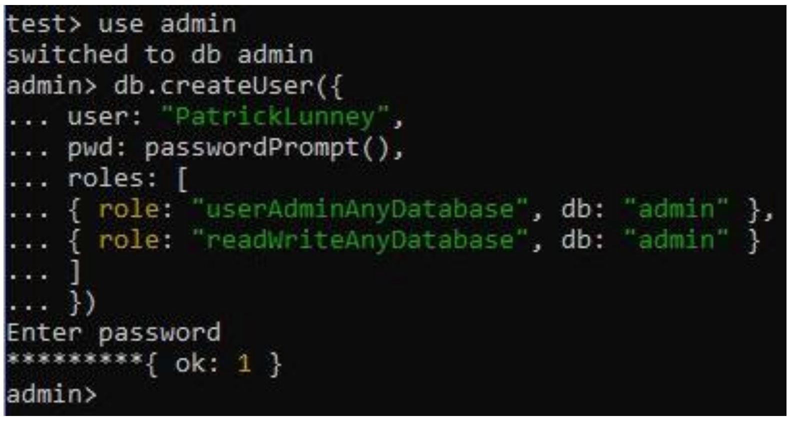
VII. Authentication Testing in MongoDB

In this section, I explore the implementation of SCRAM authentication in MongoDB and assess its effectiveness through a series of targeted tests. The aim is to evaluate how well this default authentication method secures user access, focusing on ease of setup, password handling, and resistance to common vulnerabilities.

A. Setting up and Implementing SCRAM Authentication:

When starting MongoDB as a service, access control is disabled by default, meaning anyone connecting to the MongoDB instance via mongosh can interact with the database without authentication. To secure the database, I needed to enable access control, and SCRAM authentication is automatically used once this is enabled [32]

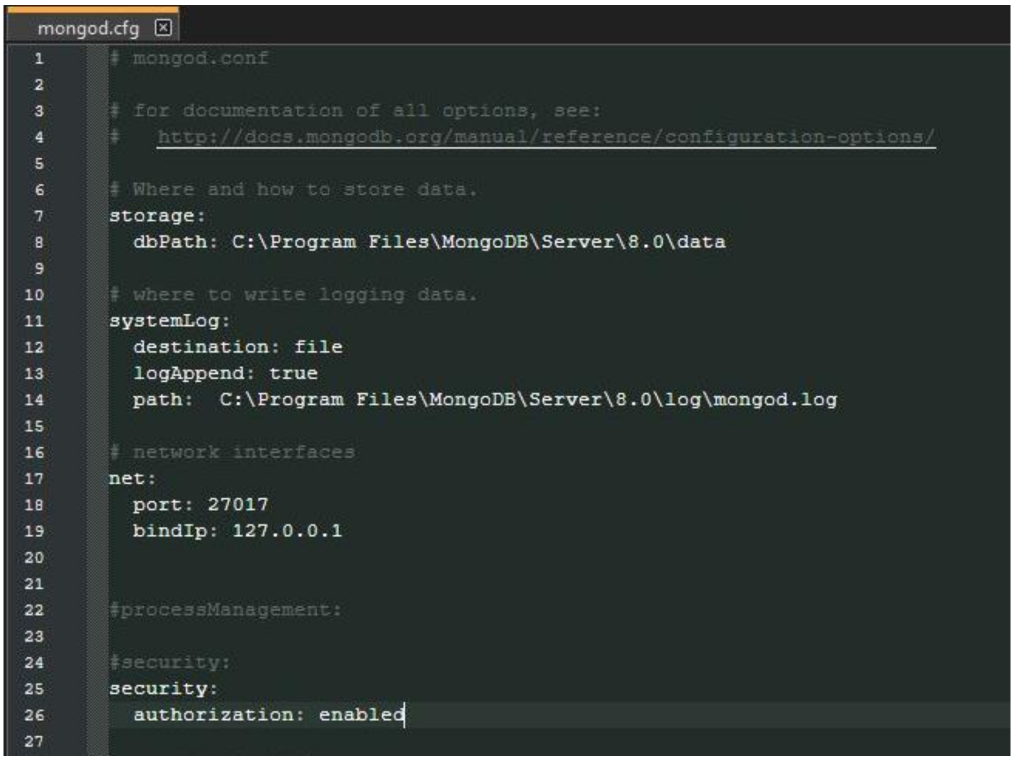
To enable access control, I first created an administrative user with the necessary privileges [33]:



```
test> use admin
switched to db admin
admin> db.createUser({
... user: "PatrickLunney",
... pwd: passwordPrompt(),
... roles: [
... { role: "userAdminAnyDatabase", db: "admin" },
... { role: "readWriteAnyDatabase", db: "admin" }
... ]
... })
Enter password
*********{ ok: 1 }
admin>
```

Fig. 23. Admin User Created in Admin Collection

This user now has full privileges to manage the database, including the ability to create new users and manage authentication. After setting up this administrative user, the next step was to shut down the service, update the configuration file, and restart the service with access control enabled [33], [34].

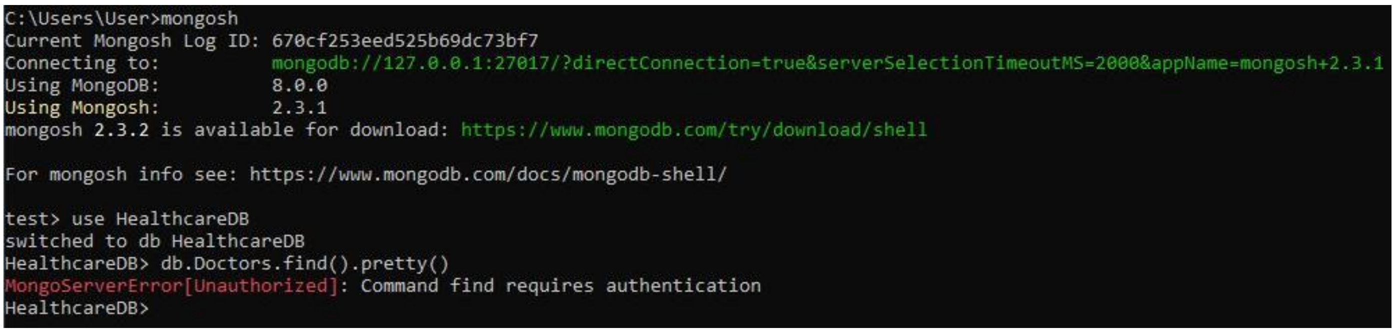


```
mongod.cfg

# mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# Where and how to store data.
storage:
  dbPath: C:\Program Files\MongoDB\Server\8.0\data

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path:  C:\Program Files\MongoDB\Server\8.0\log\mongod.log

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1


#processManagement:

#security:
security:
  authorization: enabled

```

Fig. 24. Editing Config file to Enable Access Control

B. *Enabling Authentication:*

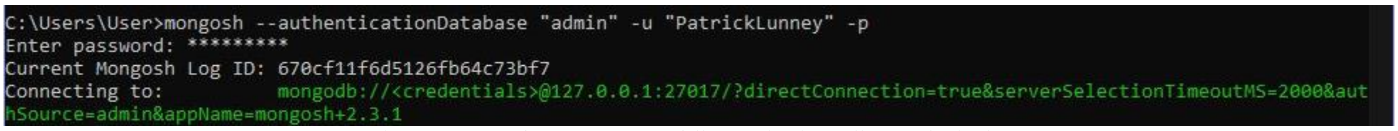
Once the service was restarted with the updated configuration, authentication was enforced. Initially, logging in through mongosh without credentials still provided access to the database. However, no operations could be performed without proper authentication, which enforces the SCRAM mechanism. This behaviour differs from MariaDB, where users are immediately prompted to authenticate.



```
C:\Users\User>mongosh
Current Mongosh Log ID: 670cf253eed525b69dc73bf7
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.1
Using MongoDB:          8.0.0
Using Mongosh:          2.3.1
mongosh 2.3.2 is available for download: https://www.mongodb.com/try/download/shell

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

test> use HealthcareDB
switched to db HealthcareDB
HealthcareDB> db.Doctors.find().pretty()
MongoServerError[Unauthorized]: Command find requires authentication
HealthcareDB>
```

Fig. 25. Mongosh Client Allowing Database Access without Asking for Credentials on Login

From here, logging in with the correct method, using the "--authenticationDatabase "database name" -u "account name" -p" allowed the account to function as intended [33].

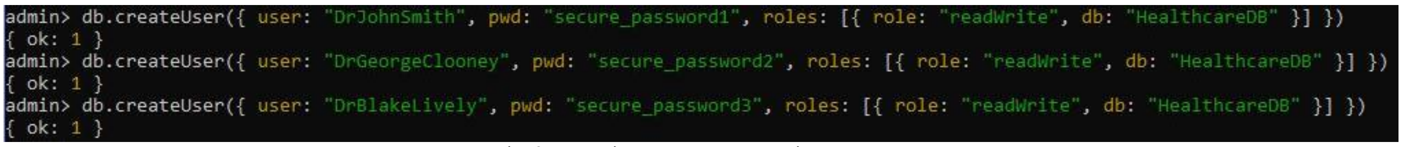


```
C:\Users\User>mongosh --authenticationDatabase "admin" -u "PatrickLunney" -p
Enter password: *********
Current Mongosh Log ID: 670cf11f6d5126fb64c73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&aut
hSource=admin&appName=mongosh+2.3.1
```

Fig. 26. Exa,[;e pf Account Successfully Logging in to allow Authorised Use

C. *Creating Users with SCRAM:*

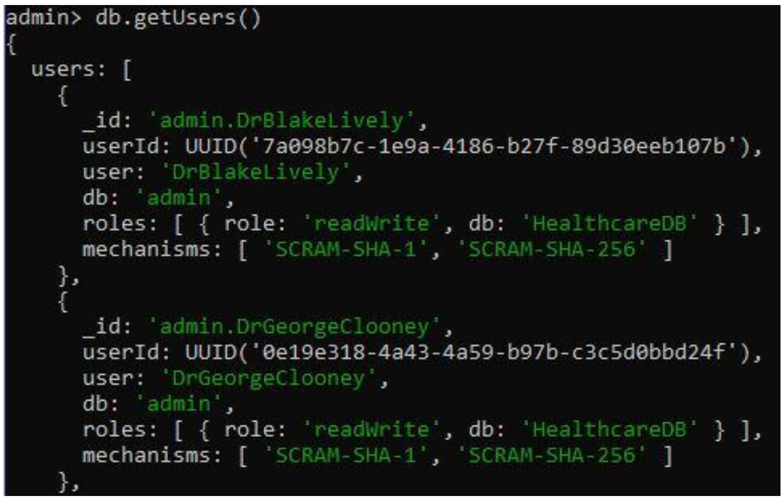
From here, I created three new user accounts for testing purposes, each with basic read-write access to the HealthcareDB [35]:



```
admin> db.createUser({ user: "DrJohnSmith", pwd: "secure_password1", roles: [{ role: "readWrite", db: "HealthcareDB" }] })
{ ok: 1 }
admin> db.createUser({ user: "DrGeorgeClooney", pwd: "secure_password2", roles: [{ role: "readWrite", db: "HealthcareDB" }] })
{ ok: 1 }
admin> db.createUser({ user: "DrBlakeLively", pwd: "secure_password3", roles: [{ role: "readWrite", db: "HealthcareDB" }] })
{ ok: 1 }
```

Fig. 27. Creating Doctor Accounts in MongoDB

To confirm that SCRAM was being used for authentication, I used the getUsers command and checked the system.users collection [36]. This provided information verifying that the newly created users were indeed using SCRAM as the default authentication mechanism.



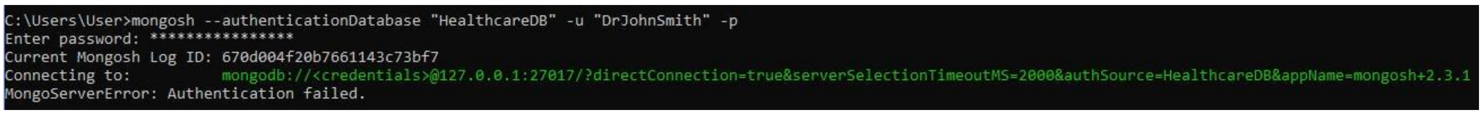
```
admin> db.getUsers()
{
  users: [
    {
      _id: 'admin.DrBlakeLively',
      userId: UUID('7a098b7c-1e9a-4186-b27f-89d30eeb107b'),
      user: 'DrBlakeLively',
      db: 'admin',
      roles: [ { role: 'readWrite', db: 'HealthcareDB' } ],
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
    },
    {
      _id: 'admin.DrGeorgeClooney',
      userId: UUID('0e19e318-4a43-4a59-b97b-c3c5d0bbd24f'),
      user: 'DrGeorgeClooney',
      db: 'admin',
      roles: [ { role: 'readWrite', db: 'HealthcareDB' } ],
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
    },
```

Fig. 28. getUsers Command in use, Showing User Account Information

D. *Authentication Tests:*

1) *Test 1: Successful Login.*

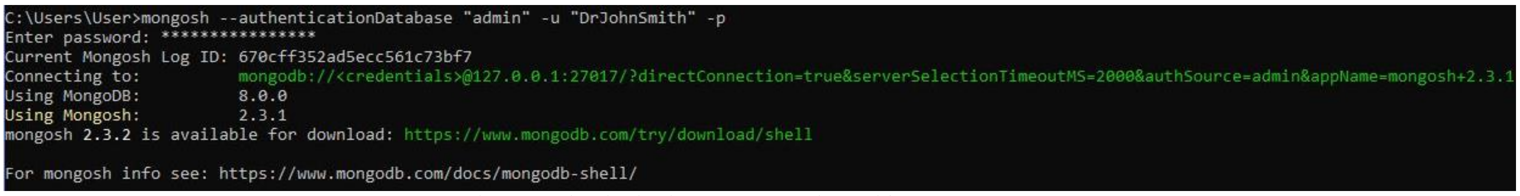
I tested logging in with the correct credentials by first attempting to authenticate into the HealthcareDB where the user privileges reside. However, MongoDB requires users to authenticate into the database where the account was created [37]. This necessitated logging in through the admin database for each account, despite the privileges being assigned to HealthcareDB.



```
C:\Users\User>mongosh --authenticationDatabase "HealthcareDB" -u "DrJohnSmith" -p
Enter password: ****************
Current Mongosh Log ID: 670d004f20b7661143c73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=HealthcareDB&appName=mongosh+2.3.1
MongoServerError: Authentication failed.
```

Fig. 29. Attempted Authentication into a Database where Account was not Created in

Once authenticated into the admin database, the privileges were limited to the roles granted in HealthcareDB, which functioned as expected.

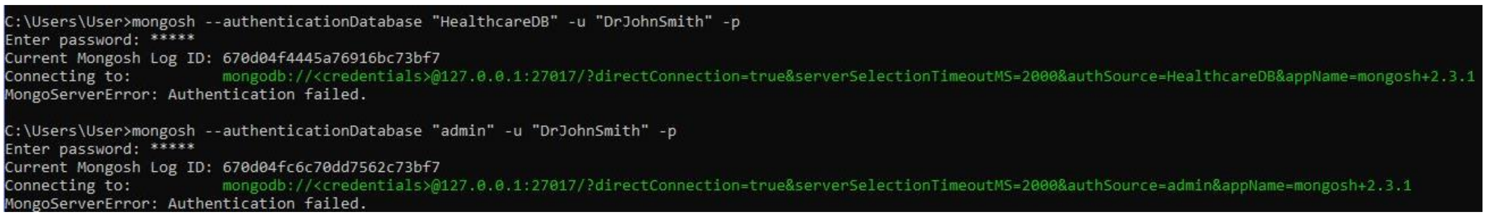


```
C:\Users\User>mongosh --authenticationDatabase "admin" -u "DrJohnSmith" -p
Enter password: ****************
Current Mongosh Log ID: 670cff352ad5ecc561c73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=admin&appName=mongosh+2.3.1
Using MongoDB:          8.0.0
Using Mongosh:          2.3.1
mongosh 2.3.2 is available for download: https://www.mongodb.com/try/download/shell

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/
```

Fig. 30. Authenticating into Correct Database with no Error Message

2) *Test 2: Invalid Password Handling*

I attempted to log in with incorrect passwords in both the admin and HealthcareDB databases. In all cases, MongoDB correctly denied access, returning an authentication failure message. This demonstrated that the SCRAM mechanism is working as intended for password validation.

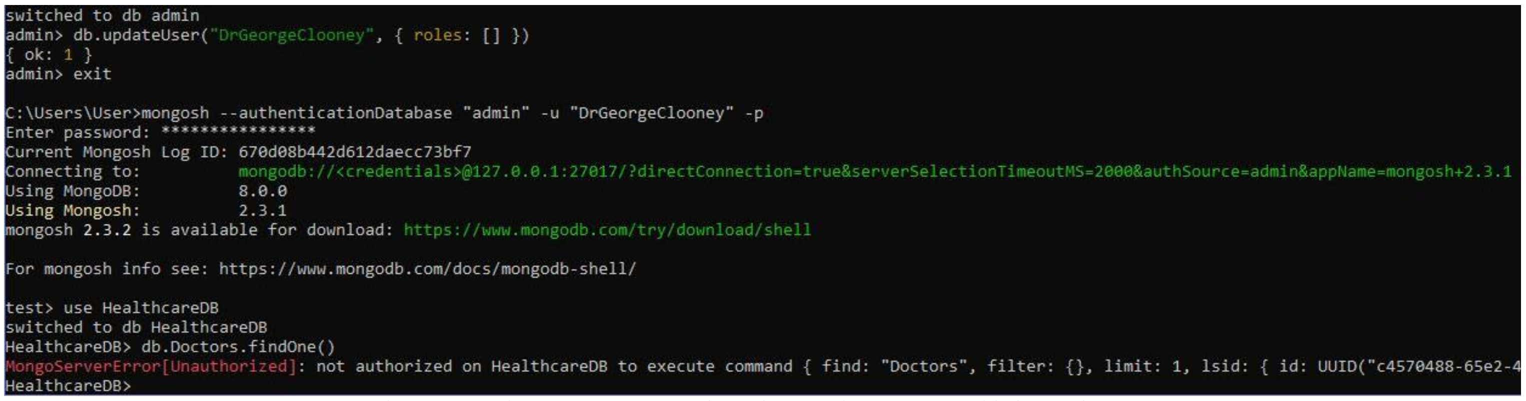


```
C:\Users\User>mongosh --authenticationDatabase "HealthcareDB" -u "DrJohnSmith" -p
Enter password: *****
Current Mongosh Log ID: 670d04f4445a76916bc73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=HealthcareDB&appName=mongosh+2.3.1
MongoServerError: Authentication failed.

C:\Users\User>mongosh --authenticationDatabase "admin" -u "DrJohnSmith" -p
Enter password: *****
Current Mongosh Log ID: 670d04fc6c70dd7562c73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=admin&appName=mongosh+2.3.1
MongoServerError: Authentication failed.
```

Fig. 31. Attempted Authentication with Incorrect Password in Both Databases

3) Test 3: Account Locking.

MongoDB's default configuration does not support account locking natively. Locking an account would require additional third-party authentication tools, such as LDAP, which are outside the scope of this test [38]. To simulate an account lock, I revoked all roles from the DrGeorgeClooney user via the db.updateUser command. While this achieved the goal of blocking access, it demonstrates a key operational difference: MariaDB provides a simple, reversible 'lock' command, whereas MongoDB's default setup requires a potentially destructive and harder-to-audit change to a user's role assignments to achieve a similar result.



```
switched to db admin
admin> db.updateUser("DrGeorgeClooney", { roles: [] })
{ ok: 1 }
admin> exit

C:\Users\User>mongosh --authenticationDatabase "admin" -u "DrGeorgeClooney" -p
Enter password: ****************
Current Mongosh Log ID: 670d08b442d612daecc73bf7
Connecting to:          mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&authSource=admin&appName=mongosh+2.3.1
Using MongoDB:          8.0.0
Using Mongosh:          2.3.1
mongosh 2.3.2 is available for download: https://www.mongodb.com/try/download/shell

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

test> use HealthcareDB
switched to db HealthcareDB
HealthcareDB> db.Doctors.findOne()
MongoServerError[Unauthorized]: not authorized on HealthcareDB to execute command { find: "Doctors", filter: {}, limit: 1, lsid: { id: UUID("c4570488-65e2-4
HealthcareDB>
```

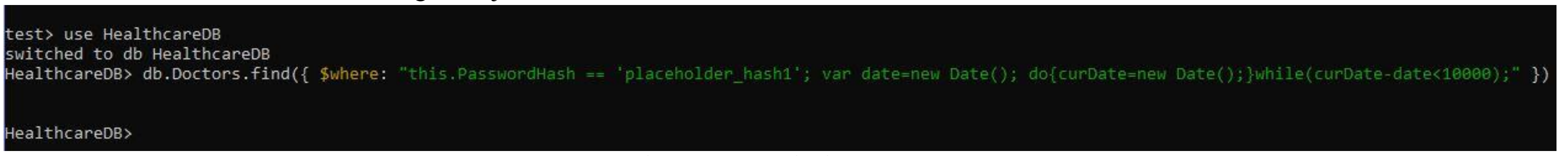
Fig. 32. Removing Doctor Account Roles and Attempting to Interact with the Database

After verifying that the user could no longer access the system, I restored the roles, and the account resumed functionality as expected.

4) Test 4: NoSQL Injection Test.

While NoSQL injection attacks differ from SQL injection – typically targeting queries instead of the authentication process – I conducted a NoSQL injection test for comparison [39].

Using the DrBlakeLively account, I authenticated into HealthcareDB and injected a malicious query intended to cause a spike in CPU usage for approximately 10 seconds [39]. However, no abnormal behaviour occurred, and MongoDB returned control to the terminal without executing the injection.



```
test> use HealthcareDB
switched to db HealthcareDB
HealthcareDB> db.Doctors.find({ $where: "this.PasswordHash == 'placeholder_hash1'; var date=new Date(); do{curDate=new Date();}while(curDate-date<10000);" })


HealthcareDB>
```

Fig. 33. NoSQL Injection Attempt

5) Test 5: Password Expiry.

MongoDB's default SCRAM authentication does not include built-in password expiry functionality. More advanced mechanisms, such as x.509 certificates, LDAP, and Kerberos, support password expiry and more robust security policies, but these require more complex setups and are beyond the scope of this basic SCRAM authentication test [40].

Therefore, I was unable to test password expiry using the default configuration, and alternative approaches, such as third-party tools or advanced MongoDB configurations, would be necessary for this feature.

VIII. Results and Discussion

It is important to recognize the broader context of security and privacy challenges within the healthcare sector, particularly in the face of growing digitalization. As discussed in [41], the increasing adoption of digital technologies, such as the Internet of Medial Things (IoMT), has heightened the need for robust security measures, exposing systems to vulnerabilities like data breaches. Their research highlights the critical privacy concerns introduced by healthcare digitalization, emphasizing the importance of evaluating security protocols in the database systems responsible for managing sensitive medical data. This perspective shapes my analysis of the authentication mechanisms in MariaDB and MongoDB, aiming to assess their ability to protect healthcare information effectively.

A. Ease of Setup in MariaDB (ed25519 implementation):

1) Installation of ed25519

The installation process for ed25519 was simple and efficient. Since the plugin is included with the MariaDB installation but not activated by default, all that was required was a single command – INSTALL SONAME 'auth_ed25519';. This one-line command activated the plugin immediately, without needing additional downloads or even a server restart. This minimal effort is advantageous in healthcare environments, where rapid deployment of secure authentication methods may be crucial.

2) Configuration

There was no need for further configuration adjustments or modifications to system variables, which contributed to the simplicity of the setup. Once installed, ed25519 worked ‘out-of-the-box’. This is beneficial when dealing with healthcare data, where minimizing configuration complexity reduces the risk of misconfigurations that could compromise sensitive information.

3) User Creation

Creating users with ed25519 mirrors standard MariaDB user creation steps with only a minor adjustment – adding ‘IDENTIFIED VIA ed25519’ to specify the authentication method. This addition felt intuitive, and anyone familiar with MariaDB’s user management would find this process straightforward.

4) Documentation/Support

The documentation was concise and easy to follow, which helped streamline the setup process. There were no issues in finding relevant instructions or encountering unclear steps. This simplicity would be valued by administrators who want a quick and effective way to enable secure authentication.

B. Ease of Setup in MongoDB (SCRAM implementation):

1) Installation of SCRAM

While SCRAM is MongoDB’s default authentication mechanism, the setup process for enabling access control added some complexity. The process involved creating an administrative user, stopping the MongoDB service, manually editing the configuration file to enable access control, and then restarting the service. This multi-step approach, while manageable, introduces a level of complexity that could be challenging in healthcare environments where secure and fast setup is needed to meet compliance deadlines or urgent patient data security demands.

2) Configuration

Modifying the configuration file to enable access control required locating the appropriate file, making manual edits, and managing system services. While not inherently difficult, this step could present a learning curve for some users who are less comfortable with NoSQL systems or manual configuration. This could be a consideration in healthcare environments, where administrators may prioritize ease of use over flexibility to ensure no misconfigurations occur that could leave sensitive health records exposed.

3) User Creation

Creating users with SCRAM was relatively straightforward, much like user creation in MariaDB. SCRAM being the default mechanism meant that no additional configurations were necessary for enabling secure authentication. MongoDB also offers an added layer of security through password prompts, which mask passwords during entry, enhancing security without complicating the process. This feature would be an added advantage in healthcare systems, providing a balance between usability and security when managing access to sensitive patient information.

4) Documentation/Support

The MongoDB documentation was comprehensive and provided the necessary steps for enabling SCRAM, although the process felt more complex due to the extra steps required for setup. While the instructions were clear, the additional steps introduced extra tasks that may be less convenient for some users. The setup, though more involved than MariaDB’s, ultimately provided strong security once completed, with resources available for guidance.

C. Authentication Tests:

1) Test 1: Successful Login.

MariaDB: All users were able to log in successfully using the correct credentials, confirming ed25519 was functioning as intended, securely verifying the password.

MongoDB: Users had to log in through the admin database where the account was created, even if their privileges were limited to HealthcareDB. The login was successful once the correct credentials were provided in the admin database.

Discussion: While both systems effectively authenticated users with valid credentials, MariaDB’s approach is more straightforward, allowing users to authenticate directly into the database where their privileges are assigned. MongoDB’s separation of the authentication database from the privileges granted in other databases introduces additional complexity. Users must authenticate into the database where the account was created (in my case, admin), even if their privileges are limited to a different database (e.g. HealthcareDB). In a healthcare environment, where rapid and clear access control is essential, this separation could slow down operations, especially in smaller, more streamlined setups.

2) Test 2: Invalid Password Handling.

MariaDB: All users were rejected when attempting to login with incorrect passwords, returning an error message for each invalid attempt, demonstrating ed25519 authentication handled the incorrect credentials.

MongoDB: Similarly rejected login attempts when incorrect passwords were provided, displaying an authentication failure message.

Discussion: Both systems handle invalid passwords as expected, preventing unauthorized access. This is particularly important in healthcare systems where unauthorized access to sensitive patient data can have severe consequences. MongoDB’s SCRAM mechanism and MariaDB’s ed25519 mechanism both enforced secure password validation without requiring additional configuration, which is a strong advantage for environments needing reliable security without complexity.

3) Test 3: Account Locking.

MariaDB: MariaDB allows manual account locking, effectively preventing users from logging in when their accounts are locked. Once the account was unlocked, users were able to log in again without issue.

MongoDB: MongoDB does not support native account locking in its default configuration. To simulate account locked, I removed user roles using the db.updateUser command, blocking access to the HealthcareDB until the roles were restored.

Discussion: MariaDB's account locking mechanism is more user-friendly and efficient, providing a built-in method to lock and unlock accounts, which could be crucial in healthcare environments where fast response to security breaches would be needed. In contrast, MongoDB requires manual role adjustments, simulating account locking through the removal of roles. This could potentially introduce delays in mitigating unauthorized access in environments where quick administrative actions are required.

MongoDB Recommendations: For healthcare environments requiring frequent account locking, MongoDB administrators may need to explore third-party solutions, or MongoDB's Enterprise Edition, which offers more advanced access control features.

4) Test 4: SQL/NoSQL Injection Test.

MariaDB: The SQL injection attempt was unsuccessful, with MariaDB rejecting the injected code as a malformed command.

MongoDB: The NoSQL injection attempt was also unsuccessful. The injected code did not cause any unusual behaviour, and MongoDB returned control to the terminal – however, the lack of a distinct error or alert for the blocked query makes real-time detection more challenging from a security monitoring perspective. A silent failure like this could easily be overlooked by a security operations team.

Discussion: Both systems demonstrated resilience against injection attacks, successfully preventing arbitrary code from being executed. In the context of healthcare, where data breaches can expose sensitive patient data, this resilience is vital. However, MongoDB's lack of a clear error message when an injection attempt is blocked could make it harder for administrators to detect these attempts, and potentially delaying security responses.

MongoDB Recommendations: Adding more explicit feedback or error messages for failed injections in MongoDB would improve monitoring and alerting for potential injection attacks, especially in healthcare environments that require immediate responses to potential breaches.

5) Test 5: Password Expiry.

MariaDB: After enabling the disconnect_on_expired_password setting, MariaDB effectively prevented users from executing commands until they reset their password. However, there was no explicit notification to users that their password had expired, which is a usability issue. Users could log in without being prompted to change their password, only realizing the password had expired when they attempted to perform an action on the database. Additionally, the lack of password history enforcement allowed users to reset their password to the same value, nullifying a key benefit of the password expiry feature.

MongoDB: MongoDB's default SCRAM authentication does not support password expiry. Advanced mechanisms such as x.509 certificates or LDAP would be required to implement this feature.

Discussion: While MariaDB offers native password expiry functionality, it requires additional configuration to provide proper user feedback and enforce stronger password policies. In healthcare environments, where regulatory compliance may demand frequent password updates and expiration notices, the absence of clear feedback could present a security gap. On the other hand, MongoDB's lack of password expiry in its default configuration presents a significant gap for healthcare organizations that depend on default setups.

MariaDB Recommendations: The absence of a clear message upon login, such as "Your password has expired," limits the user experience and security. Implementing a prompt that forces the user to reset their password immediately upon login would improve this. On top of this, allowing users to reset their password to the same value weakens security. Enforcing stronger password policies through plugins such as simple_password_check or cracklib_password_check is recommended.

MongoDB Recommendations: To match MariaDB's password expiry functionality, administrators would need to integrate advanced authentication tools such as x.509 or LDAP. Implementing native password expiry in MongoDB's default SCRAM authentication would enhance security in healthcare settings, reducing the need for external tools and easing compliance.

IX. Conclusion

This analysis explored and compared the authentication mechanisms of MariaDB and MongoDB, with a focus on security within the context of managing healthcare data. MariaDB's ed25519 authentication stood out for its ease of setup and offered robust security features, such as built-in account locking and password expiry functionality. However, the limited user feedback when passwords expire reduces its effectiveness in real-world scenarios. Despite this, MariaDB provides a user-friendly solution with strong default security, making it well-suited for healthcare environments that prioritize straightforward configuration and administrative control.

In contrast, MongoDB's SCRAM authentication is designed for flexibility, making it a strong contender for larger or more complex healthcare systems where scalability and multi-database environments are essential. However, this flexibility introduces additional complexity in managing user access and security, particularly with the separation of authentication and operational databases. MongoDB lacks native features like password expiry and account locking, which necessitates external tools or custom configurations to meet the stricter security requirements often demanded in healthcare.

In summary, MariaDB offers a more straightforward, built-in approach to database security, while MongoDB excels in scalability and flexibility but requires more configuration to achieve certain security standards. Ultimately, the choice presents a classic trade-off. For healthcare applications prioritising out-of-the-box security and simplified administration, MariaDB's model is a stronger initial choice. For large-scale environments where flexibility and scalability are paramount, MongoDB is superior, but organisations must be prepared to invest in additional configuration and tooling to meet stringent healthcare security requirements.

REFERENCES

[1] Office of the Australian Information Commissioner, "Notifiable data breaches report: January to June 2024," Sep. 16, 2024. [Online]. Available: https://www.oaic.gov.au/privacy/notifiable-data-breaches/notifiable-data-breaches-publications/notifiable-data-breaches-report-january-to-june-2024.

[2] IBM, "What is database security?" 2024. [Online]. Available: https://www.ibm.com/topics/database-security.

[3] S. Houghton, "Importance of data security in 2024," AZTech IT, Sep. 4, 2024. [Online]. Available: https://www.aztechit.co.uk/blog/importance-of-data-security.

[4] Microsoft, "SQL Server security best practices," Mar. 1, 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-server-security-best-practices.

[5] A. Paliwal, "NoSQL security: A business necessity," SecOps Solution, Nov. 18, 2023. [Online]. Available: https://www.secopsolution.com/blog/nosql-security-a-business-necessity.

[6] M. Rouse, "MariaDB," Techopedia, Feb. 16, 2024. [Online]. Available: https://www.techopedia.com/definition/mariadb.

[7] A. Pomponio, "MariaDB overview: Key features, benefits, and FAQ," OpenLogic, Jul. 29, 2021. [Online]. Available: https://www.openlogic.com/blog/mariadb-overview.

[8] MongoDB, "MariaDB vs MongoDB: Comparing the differences," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/resources/compare/mariadb-vs-mongodb.

[9] J. R. Vacca, Computer and Information Security Handbook, 3rd ed. Burlington, MA, USA: Elsevier Science & Technology, 2017. [Online]. Available: http://ebookcentral.proquest.com/lib/swin/detail.action?docID=4858374.

[10] J. O. Abimbola and O. A. Festus, "Comparative Security Vulnerability Analysis of NoSQL and SQL Database Using MongoDB and MariaDB," International Journal of Computer Trends and Technology (IJCTT), vol. 67, no. 10, pp. 20–24, Oct. 2019. doi: 10.14445/22312803/IJCTT-V67I10P104.

[11] MariaDB, "CREATE USER," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/create-user/.

[12] MariaDB, "Pluggable Authentication Overview," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/pluggable-authentication-overview/.

[13] MongoDB, "Security checklist for self-managed deployments," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/administration/security-checklist/.

[14] Satori Cyber, "11 MongoDB security features and best practices," Satori, 2024. [Online]. Available: https://satoricyber.com/mongodb-security/11-mongodb-security-features-and-best-practices

[15] MongoDB, "MongoDB security architecture," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/resources/products/capabilities/mongodb-security-architecture.

[16] Imperva, "NoSQL Injection," Imperva, 2024. [Online]. Available: https://www.imperva.com/learn/application-security/nosql-injection.

[17] MariaDB, "CREATE DATABASE," MariaDB2024. [Online]. Available: https://mariadb.com/kb/en/create-database/.

[18] MariaDB, "CREATE TABLE," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/create-table/.

[19] MongoDB, "Install mongosh," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/mongodb-shell/install/.

[20] MongoDB, "SQL to MongoDB Mapping Chart," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/reference/sql-comparison/.

[21] MongoDB, "Databases and Collections in MongoDB," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/core/databases-and-collections/.

[22] MongoDB, "db.createCollection() method," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/reference/method/db.createCollection/.

[23] MongoDB, "mongosh Methods," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/reference/method/.

[24] MariaDB, "Authentication Plugin - ed25519," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/authentication-plugin-ed25519/.

[25] MariaDB, "Account Locking," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/account-locking/.

[26] MariaDB Foundation, "Account Locking and Password Expiry in MariaDB since 10.4," YouTube, Apr. 23, 2020. [Online]. Available: https://www.youtube.com/watch?v=s3_4p4s5i7w.

[27] OWASP, "Testing for SQL Injection," OWASP Web Security Testing Guide, 2024. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.

[28] MariaDB, "User Password Expiry," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/user-password-expiry/.

[29] MariaDB, "Server System Variables," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/server-system-variables/#disconnect_on_expired_password.

[30] MariaDB, "SET PASSWORD," MariaDB, 2024. [Online]. Available: https://mariadb.com/kb/en/set-password/.

[31] MariaDB, "GRANT," MariaD, 2024. [Online]. Available: https://mariadb.com/kb/en/grant/.

[32] MongoDB, "Enable Access Control," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/v4.4/tutorial/enable-authentication.

[33] MongoDB, "Use SCRAM to Authenticate Clients on Self-Managed Deployments," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/tutorial/configure-scram-client-authentication.

[34] MongoDB, "Self-Managed Configuration File Options," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/reference/configuration-options.

[35] MongoDB, "db.createUser()" MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/reference/method/db.createUser.

[36] MongoDB, "Manage Users and Roles on Self-Managed Deployments," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/tutorial/manage-users-and-roles/.

[37] MongoDB, "Users in Self-Managed Deployments," MongoDB, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/core/security-users/.

[38] H. de A. Souza, "How I lock one user in MongoDB," MongoDB Developer Community Forums, Aug. 2022. [Online]. Available: https://www.mongodb.com/community/forums/t/how-i-lock-one-user-in-mongodb/182356/.

[39] OWASP, "Testing for NoSQL Injection," OWASP Web Security Testing Guide, 2022. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection.

[40] G. Banaag, "Set user password expiry every 30 days," MongoDB Developer Community Forums, Feb. 2020. [Online]. Available: https://www.mongodb.com/community/forums/t/set-user-password-expiry-every-30-days/306.

[41] M. Paul, L. Maglaras, M. A. Ferrag, and I. Almomani, "Digitization of healthcare sector: A study on privacy and security concerns," ICT Express, vol. 9, no. 4, pp. 571–588, Aug. 2023. doi: 10.1016/j.icte.2023.02.007.