MDB100

Security

MongoDB Database and Security

1

Release: 20250221

Topics we cover

- Authentication mechanisms
- Authorization
- Roles
- Encryption
- Auditing
- Additional security measures

MongoDB Security

History of MongoDB security

Before 2017, installing MongoDB on AWS was only:

```
sudo yum install mongodb
```

Can anyone see the problem this causes?

MongoDB has always had security - we just didn't enforce its use - which meant a lot of people never thought about it.
Now MongoDB makes us at least think about it and warns us if we are insecure.
MongoDB Atlas is always secure.

Security in MongoDB

- Authentication
- Authorization
- Views
- Transport Encryption
- Encryption-at-Rest
- Field-Level Encryption
- Auditing
- Log Redaction
- Network Restrictions

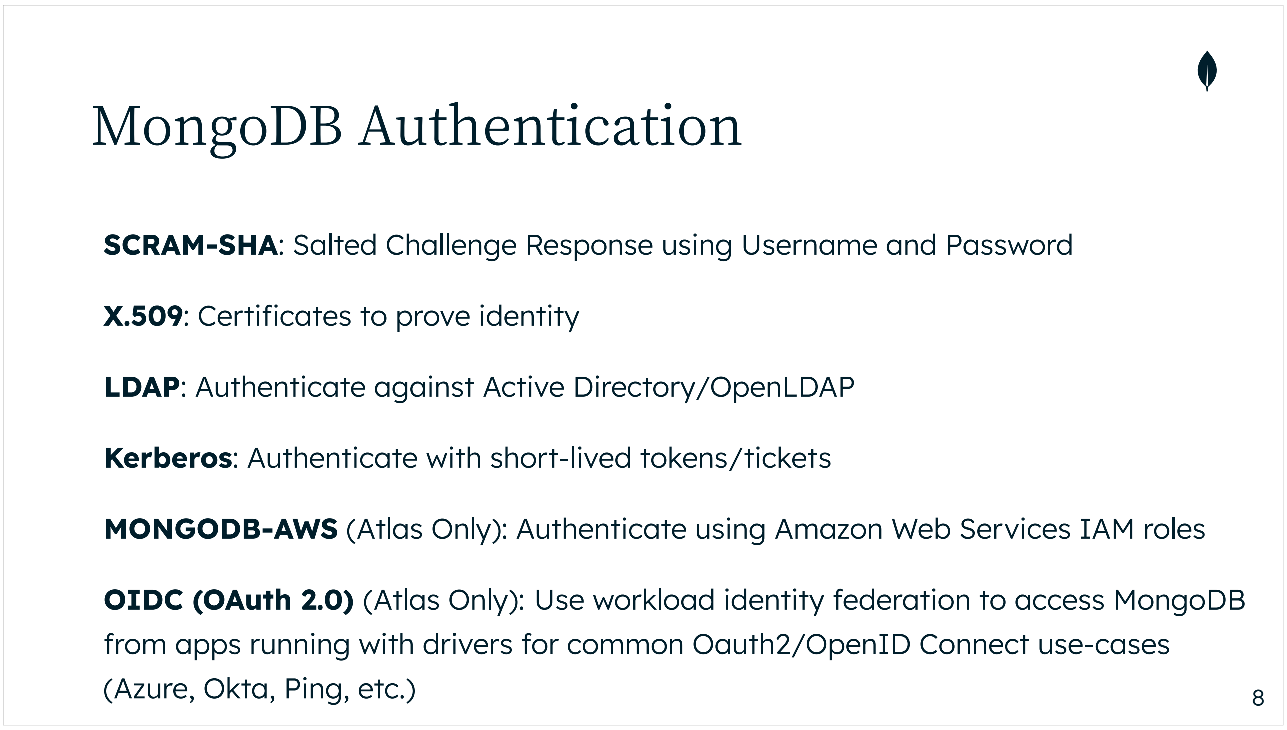
There are different concepts of security to understand with MDB

Authentication

Authentication is proving who you are

By proving one or more factors:

- Something you know - like a password
- Something you have - like a certificate
- Something you are - like biometrics, or an incoming IP address



MongoDB Authentication

SCRAM-SHA: Salted Challenge Response using Username and Password

X.509: Certificates to prove identity

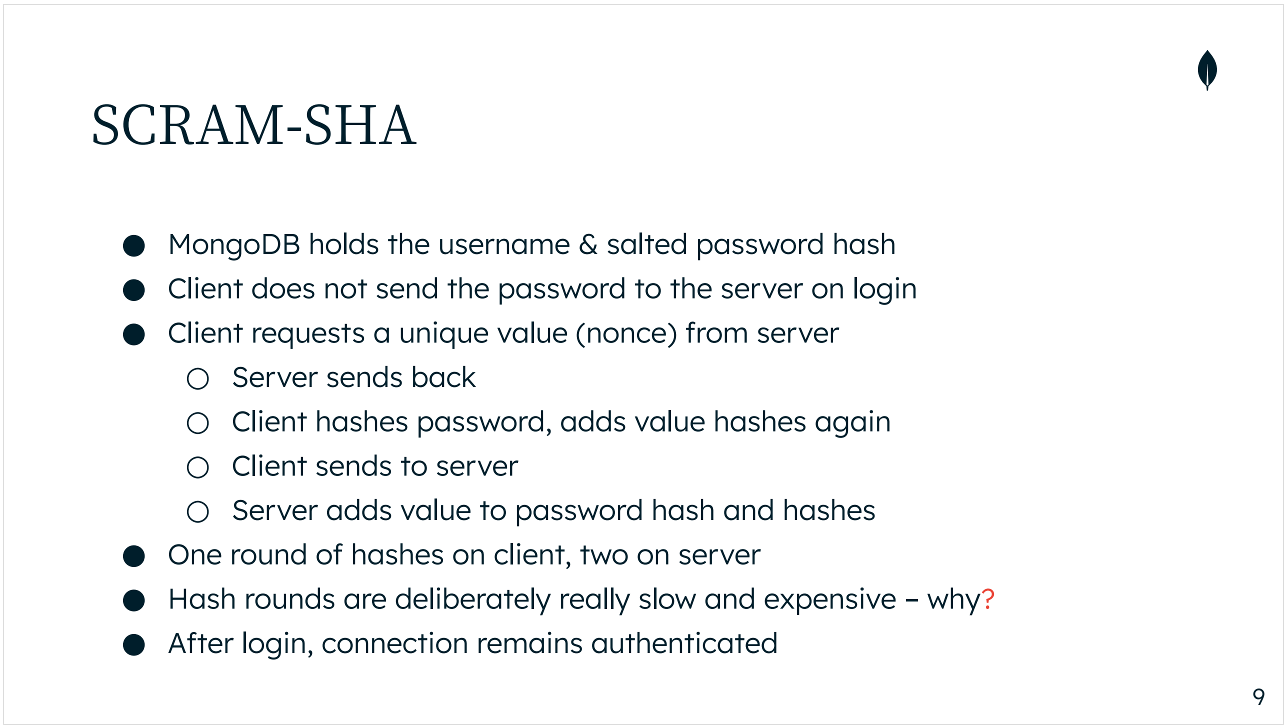
LDAP: Authenticate against Active Directory/OpenLDAP

Kerberos: Authenticate with short-lived tokens/tickets

MONGODB-AWS (Atlas Only): Authenticate using Amazon Web Services IAM roles

OIDC (OAuth 2.0) (Atlas Only): Use workload identity federation to access MongoDB from apps running with drivers for common Oauth2/OpenID Connect use-cases (Azure, Okta, Ping, etc.)

MongoDB supports multiple authentication mechanisms - the main ones are listed here.
None of these are really appropriate for end-users of an application or website - database users should be for admins and services.



SCRAM-SHA

- MongoDB holds the username & salted password hash
- Client does not send the password to the server on login
- Client requests a unique value (nonce) from server
  - Server sends back
  - Client hashes password, adds value hashes again
  - Client sends to server
  - Server adds value to password hash and hashes
- One round of hashes on client, two on server
- Hash rounds are deliberately really slow and expensive – why?
- After login, connection remains authenticated

A round of hashes is 10 or 15 thousand hashes, logins are deliberately slow and CPU intensive to avoid brute force attacks.
This is important to know, we must only log in when we have to - create a single MongoClient object in applications not one per database call or web service call.
MongoClient has a connection pool and is thread safe.

SCRAM-SHA – Pros and Cons

Pros

- Simple
- Easy to understand
- Secure

Cons

- Manage separately for each database cluster
- Authentication is local (added responsibility)
- What if the passwords become known by people?
- How to secure passwords for service users (i.e. the application)?

X.509 Certificate

X.509 Certificate presented by Client (as part of TLS connection)

- MongoDB checks user DB to see if it knows and trusts that certificate.
- The certificate is the only portion of the PEM file that is served to clients.
- The private key is never provided to any other service across the network.
- Private key may have a password too long to be typed in on client
- Sometimes the certificate needs to be in a client end key store

When logging into MongoDB via X.509 we need a PEM file that includes both the certificate and the private key, similar to what the MongoD instances use.

The private key and certificate are in the single PEM file. The certificate is the only portion of this file that is served to clients as part of a TLS connection or as part of the intra-cluster authentication. A randomly generated value is provided to clients as part of the TLS connections.

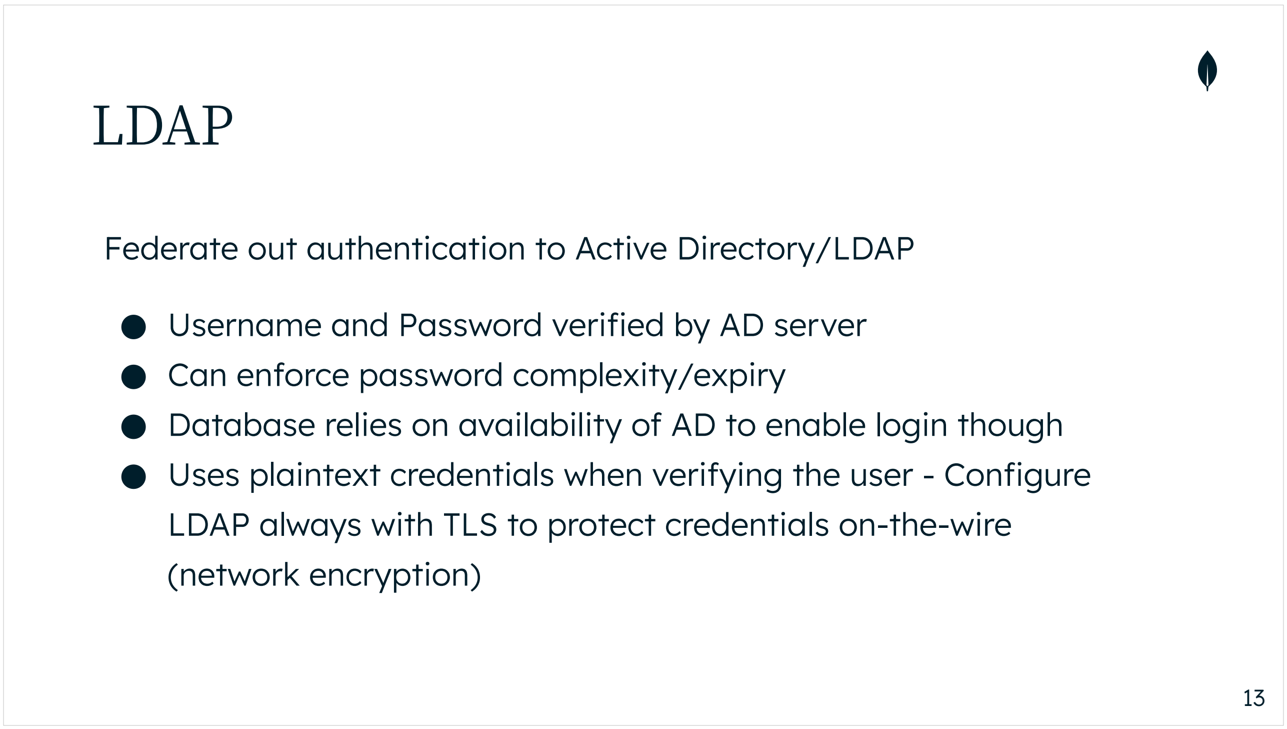
X.509 Certificate – Pros and Cons

Pros

- Can centrally provision credentials
- Can have a revocation list (revoked certificates) or OCSP

Cons

- Need a separate mechanism for Authorization like LDAP or Local
- Certificates need a password to be secure - Still need a password store



LDAP

Federate out authentication to Active Directory/LDAP

- Username and Password verified by AD server
- Can enforce password complexity/expiry
- Database relies on availability of AD to enable login though
- Uses plaintext credentials when verifying the user - Configure LDAP always with TLS to protect credentials on-the-wire (network encryption)

LDAP Authentication has has been depreciated in MongoDB Version 8.0. It will be fully removed in a future version.

LDAP - Pros and Cons

Pros

- Common in organizations
- Allows users to log in with the same password they use for their desktop
- Single mechanism for Authentication and Authorization
- Easy to add or remove users from all databases

Cons

- Relies on the AD server being up
- Supports alternate servers
- Configuration requires an understanding of the LDAP Tree
- Requires TLS connection to LDAP to be secure

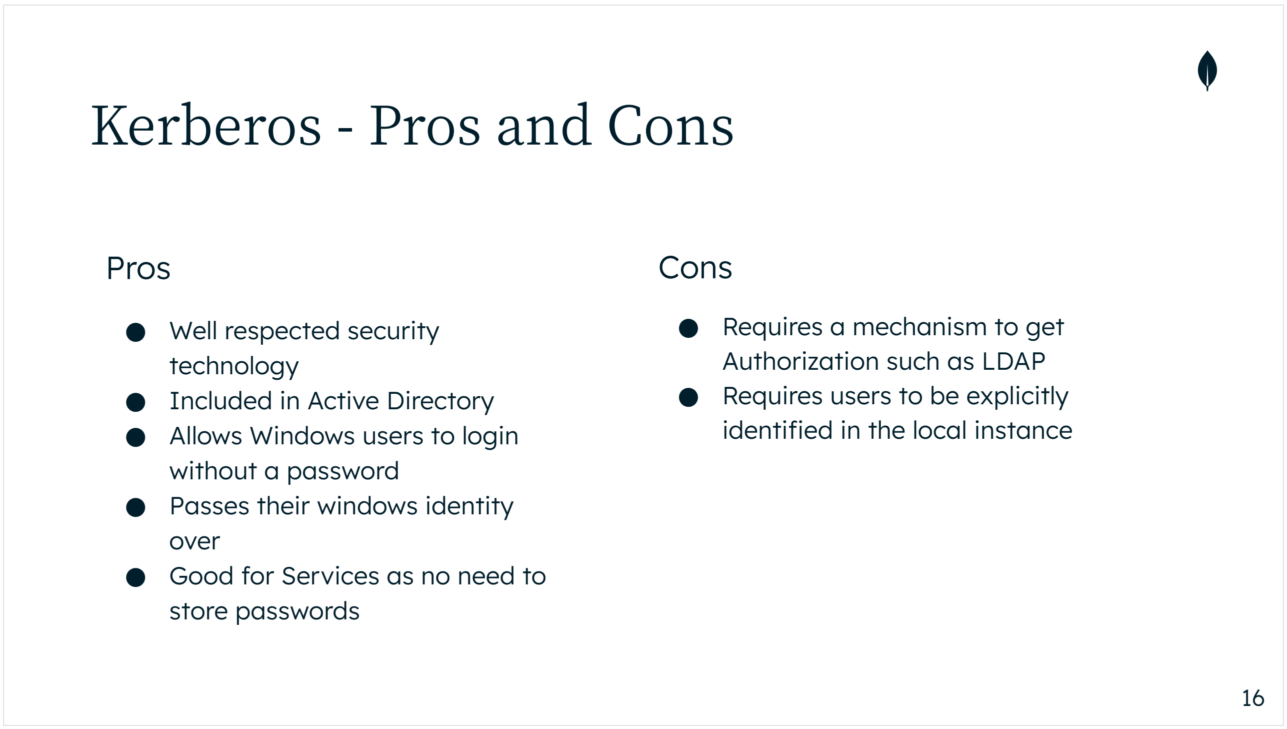
Kerberos

Well regarded and proven

Similar to X.509

- However new tickets (certs) are requested frequently
- Users log in to a Kerberos server to get a ticket that is used to log into MongoDB*
- Needs Kerberos Infrastructure set up to use it

*This is simplified description - a full explanation of Kerberos is out of the scope of this training.



Kerberos - Pros and Cons

Pros

- Well respected security technology
- Included in Active Directory
- Allows Windows users to login without a password
- Passes their windows identity over
- Good for Services as no need to store passwords

Cons

- Requires a mechanism to get Authorization such as LDAP
- Requires users to be explicitly identified in the local instance

With Kerberos we can also use MongoDB’s local Role-based access control.

OpenID Connect OIDC (Auth 2.0)

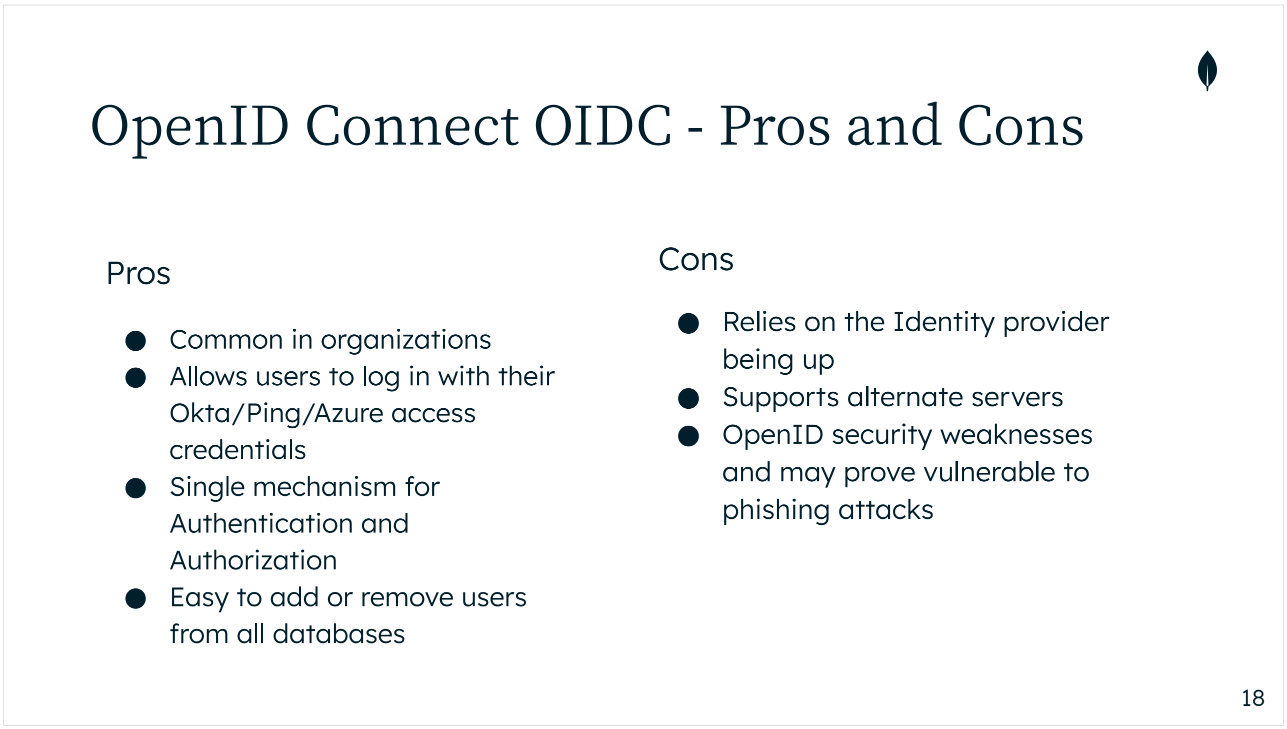
Federate workforce authentication to Okta/ Azure/Ping (or any other cloud provider that supports this authentication)

- Username and Password verified by the external provider
- Can enforce password complexity/expiry
- Database relies on availability of Identity provider to enable login though
- Enforce policies to minimize the risk of long-living DB connections
- Complete control over user lifecycle management

Available from Atlas v7.0 RC
OpenID Connect is a modern and open authentication protocol built on the OAuth 2.0 framework. This protocol is agnostic to a cloud provider. Any identity provider such as Okta, Azure AD, or Ping Identity that supports OIDC can be configured in Atlas for workforce authentication and authorization to MongoDB Atlas deployments.

Workforce Identity Federation allows employees/human user to access MongoDB clusters with Single Sign On using their corporate IdP such as Azure Entra ID, Okta, Ping etc. Customers have full control on authentication policies (e.g. password complexity/rotation, MFA, trusted device/network check etc.) and user lifecycle through their IdP. It also simplifies access management as you can authorize users with IdP groups. We never see or store any secrets. If your customers currently use LDAP, offer them to switch to this new mechanism. Currently supported by MongoDB Shell and Compass.

Workload Identity Federation allows applications to access MongoDB Clusters with Azure Service Principals (including Managed Identities) and GCP Service Accounts. Indeed this mechanism allows you to access clusters with an OAuth2.0 token, so any authorization service supporting OAuth2.0 can be used to access clusters. Again, We never see or store any secrets. Customers have full control on secrets and secret rotation per their existing compliant environment. Further more, they they can also get rid of secrets completely, because it allows passwordless authentication for apps running on certain resources (as long as the cloud provider supports) such as Azure VM, Google Compute Engine, AKS, GKE etc. It is currently supported by Python, Java and Node drivers, and soon will be available in C# and Go drivers.



OpenID Connect OIDC - Pros and Cons

Pros

- Common in organizations
- Allows users to log in with their Okta/Ping/Azure access credentials
- Single mechanism for Authentication and Authorization
- Easy to add or remove users from all databases

Cons

- Relies on the Identity provider being up
- Supports alternate servers
- OpenID security weaknesses and may prove vulnerable to phishing attacks

Read more: https://www.mongodb.com/docs/atlas/security-oidc/

What to use and when?

Human users

- Should always have individual logins
- Centralized Identity Management (LDAP/Kerberos/OIDC) is preferable
- SCRAM-SHA is secure but not centralized, requires more maintenance
- X.509 needs certificates with passwords and either local or LDAP Authorization

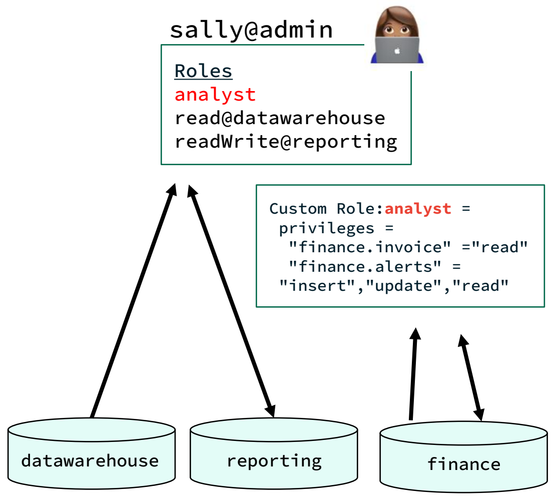
Software/Service users

- No human should know the credentials used by a service
- Kerberos is a great option for Windows Services
- On Atlas with AWS, IAM is available.
- On Atlas with Azure or Okta, OIDC is available.
- Can generate a random password (and associated user) at install time
- Store passwords securely at the application end (i.e, in an Enterprise Password MS)

Authorization (AUTHZ)

Authorization is what you are allowed to do.

- Users defined by username@database
- Database is **admin** by default
- The **system.users** collection in the admin database stores authentication and authorization details
- Users have 1 or more Roles
  - Built-in roles or custom roles
  - Roles can 'inherit' from others
  - Database roles or Atlas roles
- Roles include 1 or more privileges - A privilege means a user can perform an actions on a resource



```
Custom Role:analyst =
 privileges =
  "finance.invoice" ="read"
  "finance.alerts" =
 "insert","update","read"
```

Historically you could create a user database anywhere in MongoDB - nowadays this is deprecated and users are by default managed in the admin database. But in some scenarios, users might be also created somewhere (i.e, "$external" database)
Regardless of where the user is created, all users reside in the admin system.users collection in the new versions.
The **system.users** collection in the **admin** database stores user authentication and authorization information.
Sally is not an admin - she works as an analyst. Sally@admin is her username because every user is @admin (With some special exceptions)
Sally analyses historic data and looks for invoices which might be problematic - like not being paid, she flags these for the finance team.
She also creates reports on what she has discovered.
She has three roles: two built-in roles allowing her to read the datawarehosue and read and write to reporting.
She also has a custom role "analyst" which lets her read invoices in the finance databases and modify just the 'alerts' collection in the finance database.

When creating a custom role you can base it on a built in role or a custom role in the same database and inherit the permissions of parent roles.

Database resources

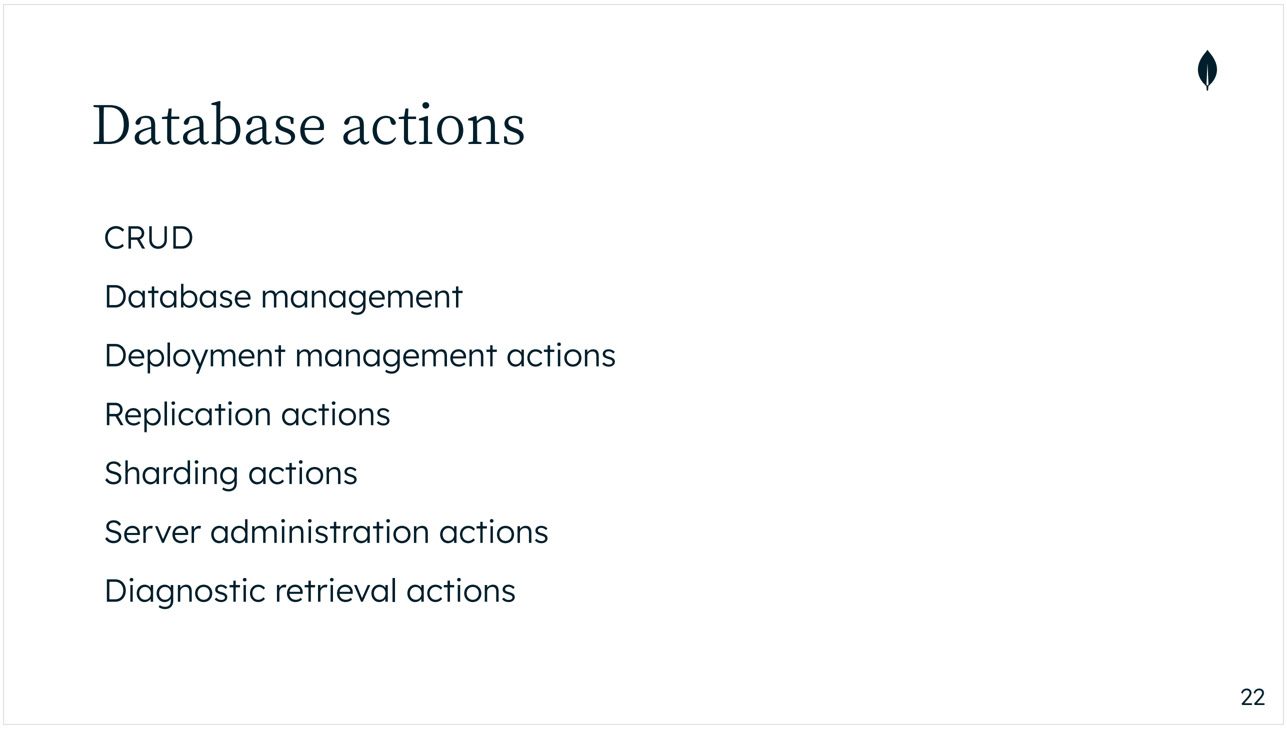
Databases

Collections

Cluster

Users

We describe the parts of the system as Resources - for example a Collection or a Cluster is an example of a Resource. Documents are not considered to be resources.



Database actions

CRUD

Database management

Deployment management actions

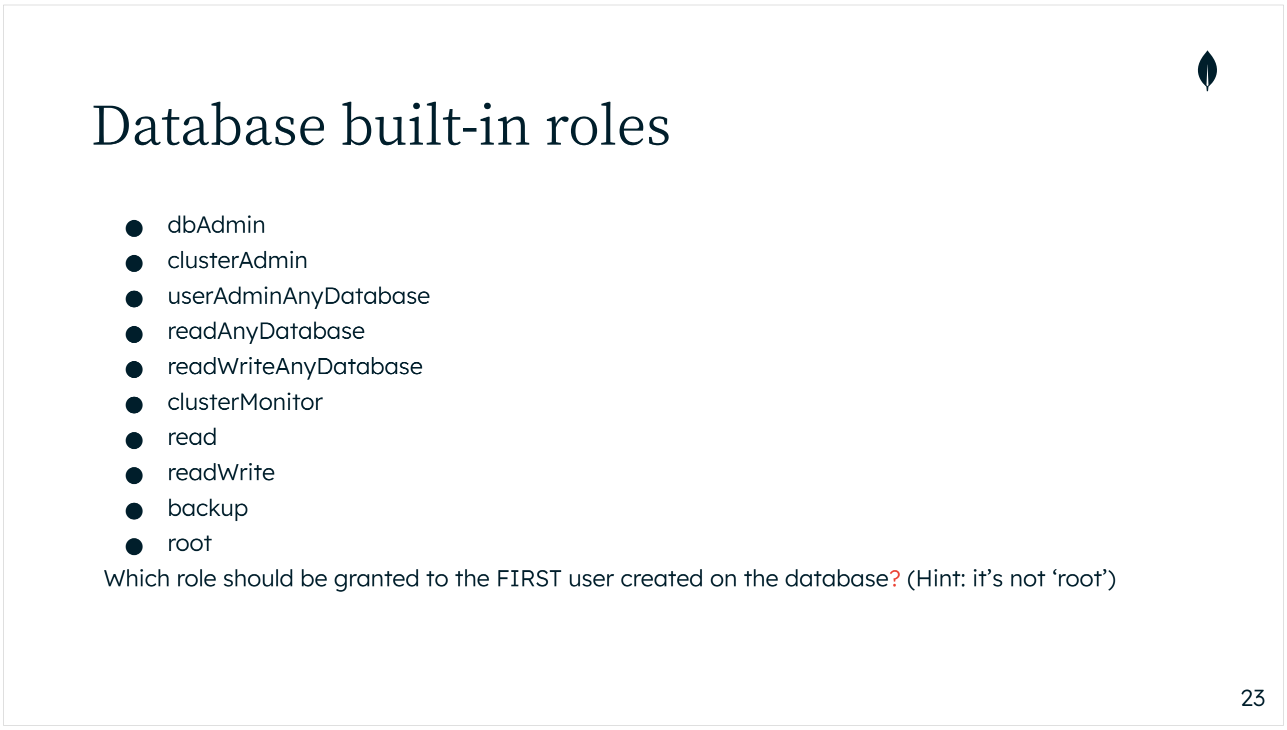
Replication actions

Sharding actions

Server administration actions

Diagnostic retrieval actions

There are many type of action - these are just categories. A Role defines what actions a user can perform on what resources.



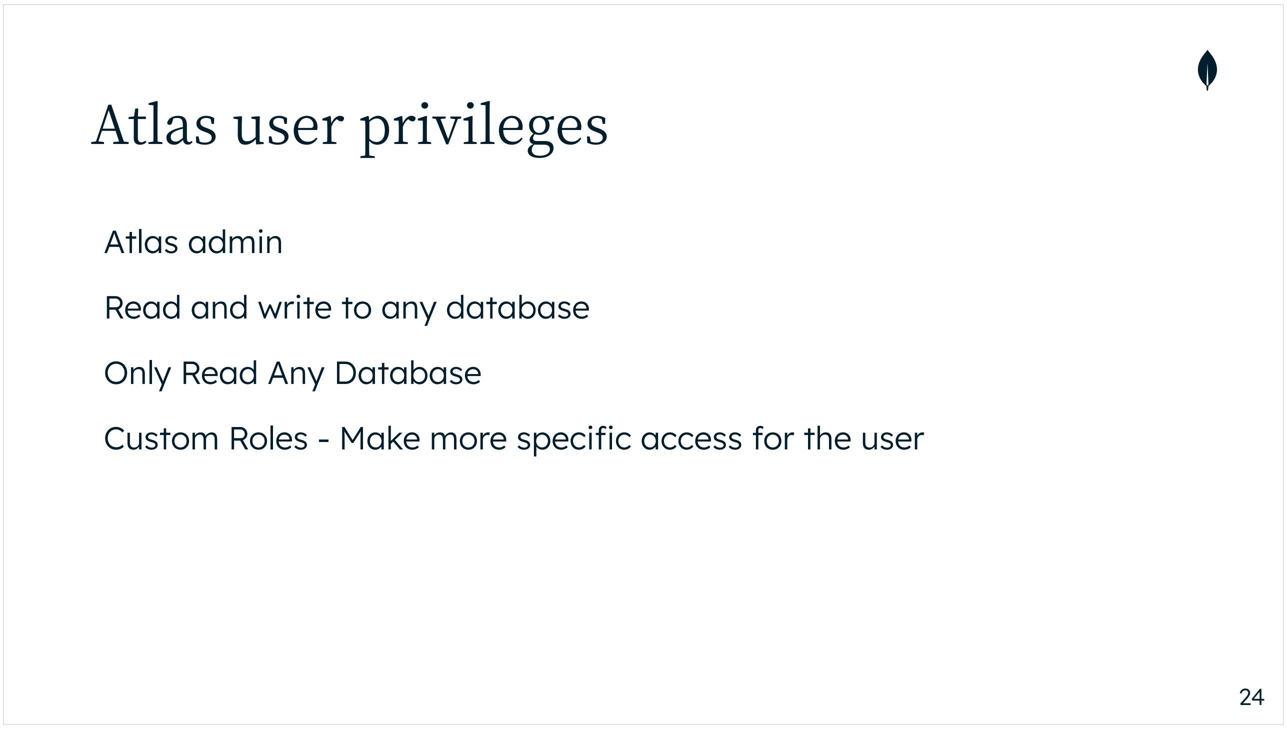
Database built-in roles

- dbAdmin
- clusterAdmin
- userAdminAnyDatabase
- readAnyDatabase
- readWriteAnyDatabase
- clusterMonitor
- read
- readWrite
- backup
- root

Which role should be granted to the FIRST user created on the database? (Hint: it's not 'root')

Note that in MongoDB we separate out the core abilities to Manage Users, Manage the Clusters, Manage Data and Read/Write data into separate roles. Our DBA role neither gives Read permissions nor Shutdown permissions for example.

Atlas users get a subset of these roles, as some admin actions cannot be performed on Atlas



Atlas user privileges

Atlas admin

Read and write to any database

Only Read Any Database

Custom Roles - Make more specific access for the user

Atlas provides a more restricted set of roles. As an example, the users cannot shut down an Atlas host or change key parameters it is running with from inside MongoDB.

Scope of Authorization

User with **root** or **__system** roles can do everything

Three types of admins - Some admin roles cannot read data!

Down to a user who can do one action on one resource

Example, read-only on a single collection

MongoDB recommend not using root but instead giving superusers a set of roles like dbAdminAnyDatabase and userAdmin. It may be worthwhile having a root user but with a password locked away for emergency access.

Avoid using use the internal __system user unless advised by MongoDB support.

The three types of admins are:

1. Data Admins
2. User Admins
3. Cluster Admins

Document Level Security Options

Roles restrict down to collection/view level- How to get obfuscation?

Application Level

- Uses a single database user
- Application code determines what a user can see and do
- Multi-tenant solutions may require more stringent security based on SLAs than a single database user can provide

Using Views

- Can create views that give a redacted view of the data
- Good choice for data-masking in APIs or BI Users

In order to get document level security, you can also use Client Side Field Level Encryption (CSFLE)

Views work well when limiting the visible fields, if we also use them to limit the visible records we need to ensure our indexes cover both the filtering and the users query.

Example on Views:
https://www.mongodb.com/docs/manual/core/views/create-view/#retrieve-medical-information-for-roles-granted-to-the-current-user

Defining Authorization

Authorization is defined in the user record - Roles are stored with the user object in *system.users*

Some authorization methods require custom roles to **map** between the user account and its permissions

- LDAP authentication without LDAP authorization requires creating a role that matches the DN (Distinguished Name)
- X509 custom users map a server's FQDN (via CN) to its authorization

Example:

```
{
   role: "CN=dba,OU=Users,DC=mongodb,DC=com",
   privileges: [],
   roles: [ "dbAdminAnyDatabase", "clusterAdmin" ]
}
```

In Atlas, you can use the GUI or an API call to set up permissions

Authorization best practices

Give minimum required rights to people - a.k.a Principle of least privilege (PoLP)

Minimum required rights to service accounts (i.e, Don't let them make indexes)

Give read-only services read-only permissions

Create roles and users in the **admin** DB only

Avoid using root/__system for normal operations

- Have a root break-glass user if needed
- Not required with Atlas as can reset passwords from the GUI

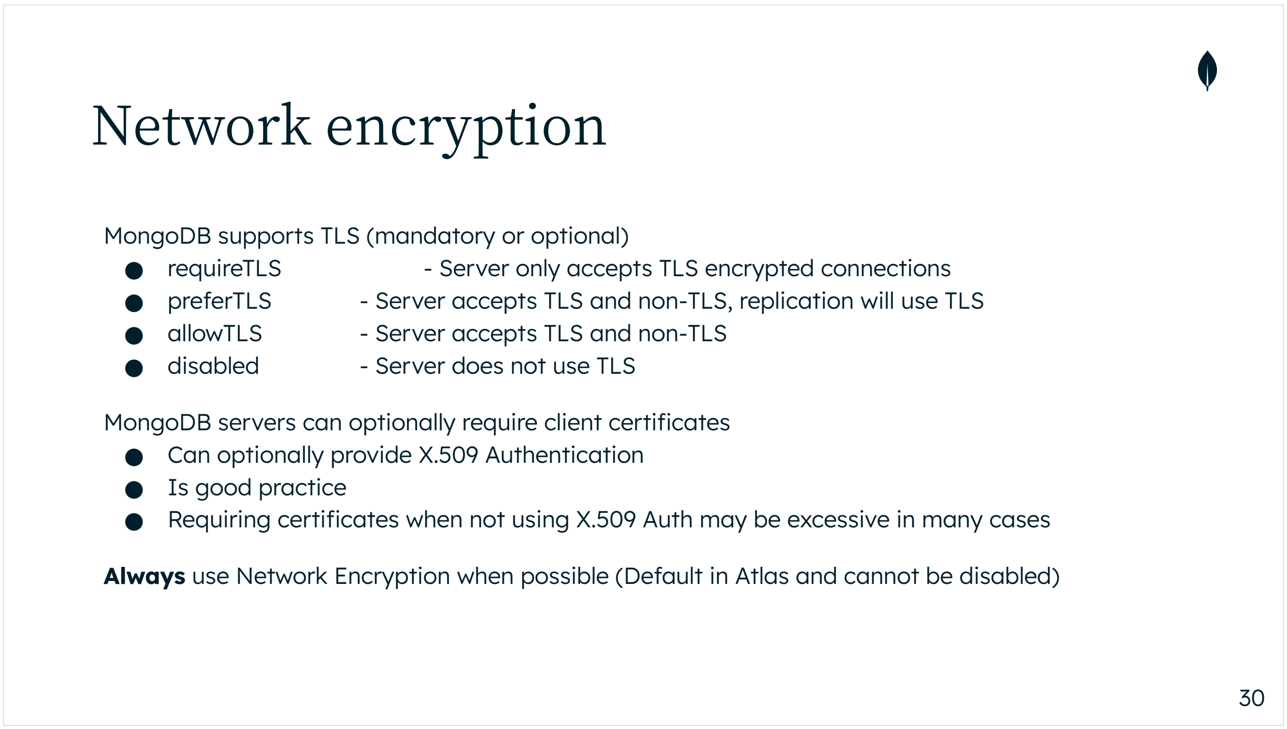
Historically users could be defined anywhere, not only in the admin DB (this feature is deprecated now)

Encryption

On the Wire (Network)

At Rest (Disk)

In Use (Client)



Network encryption

MongoDB supports TLS (mandatory or optional)

- requireTLS - Server only accepts TLS encrypted connections
- preferTLS - Server accepts TLS and non-TLS, replication will use TLS
- allowTLS - Server accepts TLS and non-TLS
- disabled - Server does not use TLS

MongoDB servers can optionally require client certificates

- Can optionally provide X.509 Authentication
- Is good practice
- Requiring certificates when not using X.509 Auth may be excessive in many cases

Always use Network Encryption when possible (Default in Atlas and cannot be disabled)

The optional TLS configuration in MongoDB allows transitioning a non-TLS Cluster to a TLS Cluster without downtime.

Encryption at Rest

With Encryption at Rest, MongoDB encrypts the files on disk

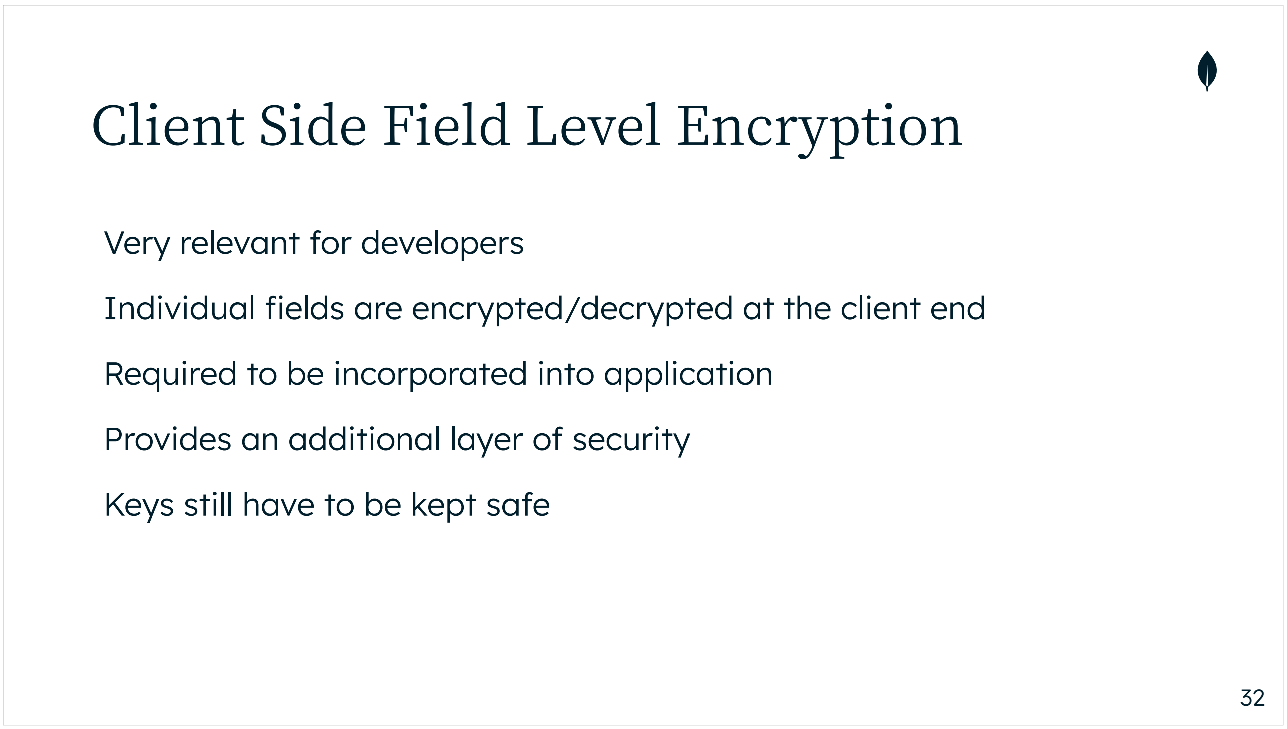
Needs a secure way to store and manage the decryption keys

- Needs an external KMIP Key Vault
- Storing Keys locally is not considered secure
- Keys must be manually rotated

Encryption at Rest protects from theft of files/disks

- Does not protect us from anyone who has gained access to the machine
- Does not protect us from theft of files if they also take the key
- Alternative to using encrypted disk volumes if we don't have them available to us

Encrypted disk volumes: full disk encryption configured at Operating System level - configuring this is out of scope for MongoDB training.



Client Side Field Level Encryption

Very relevant for developers

Individual fields are encrypted/decrypted at the client end

Required to be incorporated into application

Provides an additional layer of security

Keys still have to be kept safe

Note: Field level encryption does not get away from the requirement to have a place to keep the keys safe.

Client Side Field Level Encryption

Encryption on client side secures against:

- Data not being encrypted in cache - dumping RAM
- DBA/Admin being able to read data in the database
- A root level hacker who restarts the database without security
- Access to data by hosting/cloud provider - for example Atlas

Dumping client RAM or a compromised client is still a security risk

Encrypt and Decrypt at the Client

Using Field Level Encryption

- Server never has cleartext data
- Server never has decryption keys
- Server side can never read unencrypted data

Client obtains key from a key management system

Client encrypts and decrypts specific fields as needed (SSN, Credit Card, Name)

Client only sends encrypted data or queries with encrypted values to server

Field Level Encryption modes

Free MongoDB Community

- Encryption function exposed in driver – encryption done manually by code record.name=encrypt(name)

Enterprise & Atlas

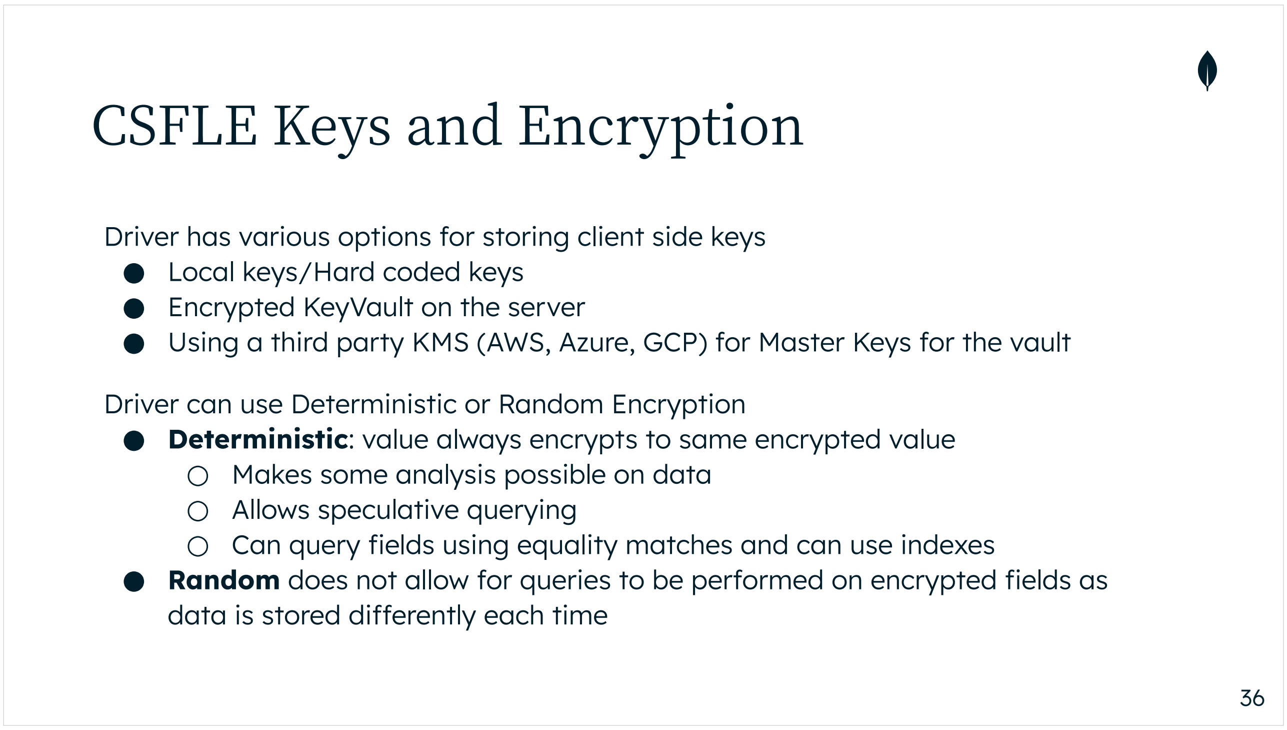
- JSONSchema defines encrypted fields
- Driver automatically encrypts CRUD operations including queries

Implementing Encryption is hard - MongoDB provides a secure and easy to use option

Automatic Encryption has some limitations

- while it can automatically encrypt a whole array, it cannot automatically encrypt fields on docs within an array.
- only available on systems where EA is supported; not supported on Alpine Linux for example.

Manual encryption is still available alongside this in Enterprise Atlas for those edge cases.



CSFLE Keys and Encryption

Driver has various options for storing client side keys

- Local keys/Hard coded keys
- Encrypted KeyVault on the server
- Using a third party KMS (AWS, Azure, GCP) for Master Keys for the vault

Driver can use Deterministic or Random Encryption

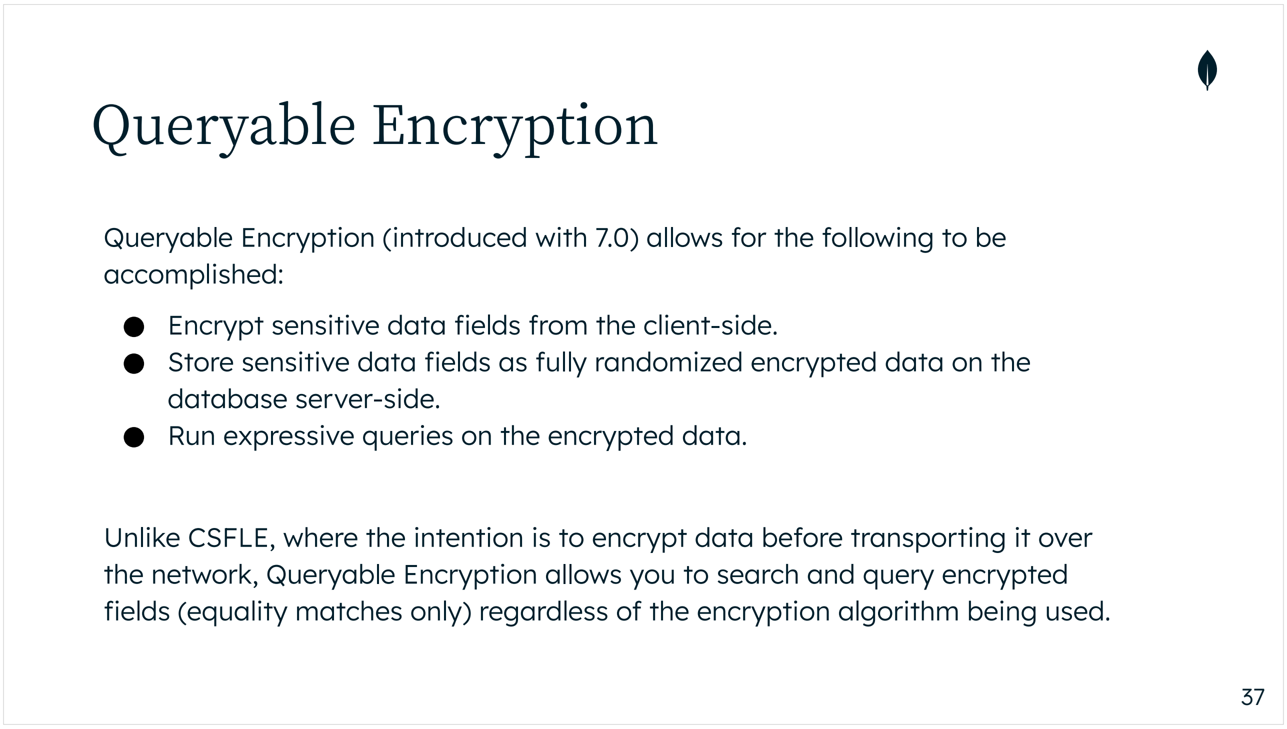
- **Deterministic**: value always encrypts to same encrypted value
  - Makes some analysis possible on data
  - Allows speculative querying
  - Can query fields using equality matches and can use indexes
- **Random** does not allow for queries to be performed on encrypted fields as data is stored differently each time

Data Encryption Keys (DEKs) are used to encrypt data on the client before sending it to the mongod server. These DEKs are stored in the keyVault on the mongod server, but are wrapped or encrypted themselves by a Customer Master Key (CMK). The CMK can be stored using a KMS provided by one of the cloud providers or a KMIP device or locally (which isn't recommended and is typically used for dev practices)

Server can store all the decryption keys, encrypted with a master key on the client so the client end only needs to find a safe way to store one key.

If a field is randomly encrypted - we cannot query it as it could have any value in the data so there is no way to turn the query term into a single value to search for.
With deterministic we encrypt the query term and search for the encrypted version which is the same everywhere.

Local mode can be used to retrieve keys from third party tools or services that are not the three Cloud Providers.
CSFLE can also use KMIP



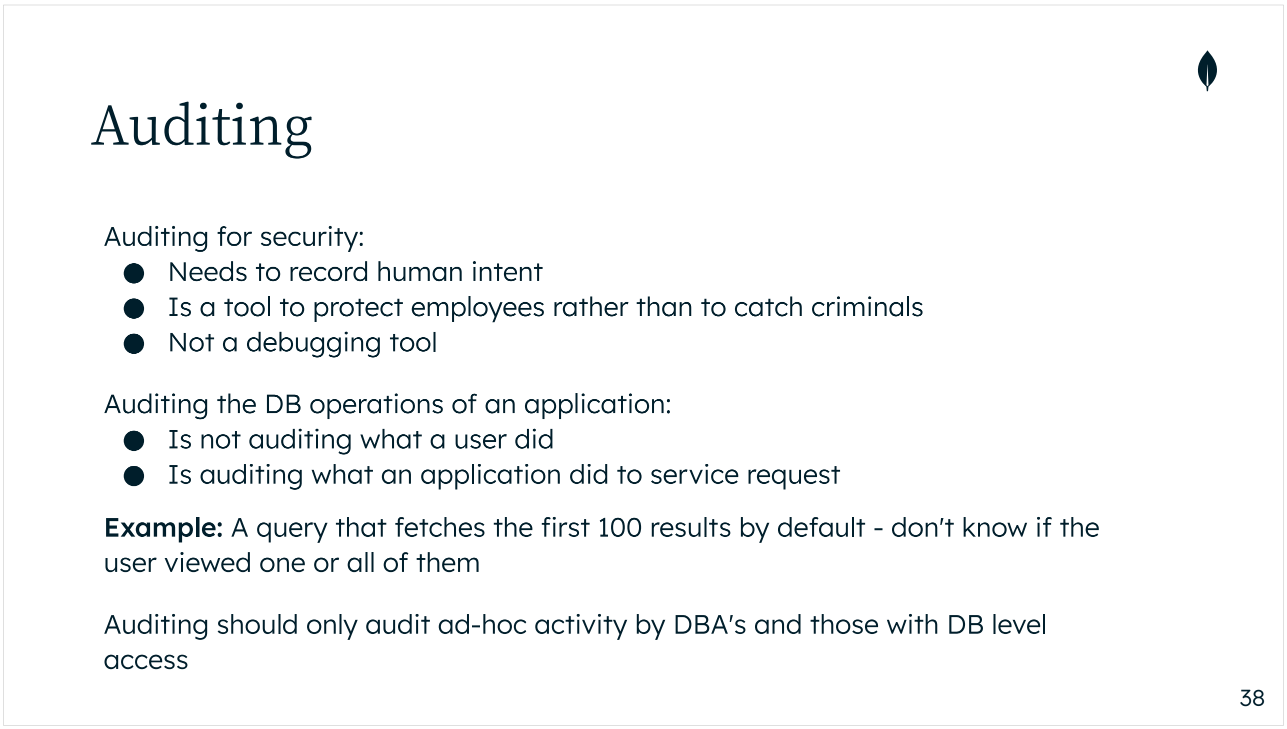
Queryable Encryption

Queryable Encryption (introduced with 7.0) allows for the following to be accomplished:

- Encrypt sensitive data fields from the client-side.
- Store sensitive data fields as fully randomized encrypted data on the database server-side.
- Run expressive queries on the encrypted data.

Unlike CSFLE, where the intention is to encrypt data before transporting it over the network, Queryable Encryption allows you to search and query encrypted fields (equality matches only) regardless of the encryption algorithm being used.

This feature can be implemented with both automatic encryption and manually via driver.



Auditing

Auditing for security:

- Needs to record human intent
- Is a tool to protect employees rather than to catch criminals
- Not a debugging tool

Auditing the DB operations of an application:

- Is not auditing what a user did
- Is auditing what an application did to service request

Example: A query that fetches the first 100 results by default - don't know if the user viewed one or all of them

Auditing should only audit ad-hoc activity by DBA's and those with DB level access

Auditing is there so in the event of an incident you can see what your employees who have direct database access did, or (important) what they did not do.
Audit is NOT a tripwire to see who is trying to perform tasks that security does not allow them to.
Auditing the behaviour of an application / service account is rarely useful, the application should audit what its users are trying to achieve.
For debugging - often the OpLog (Database Transaction Log) is a good place to see what actually happened.

Security as a whole should be handled using **Security Information and Event Management (SIEM)** tools and services offering a holistic view of an organization's information security. SIEM tools provide: Real-time visibility across an organization's information security systems not just MongoDB - The Audit trail may flow into that.

MongoDB Auditing capability

- Can record all successful or prevented operations
- Only audits where security prevents a non-CRUD activity by default
  - Why audit where security stopped a thing? Is that useful?
- Auditing flushes the disk before each audited operation
  - Only audit valuable ad-hoc information
  - Do not audit 'Business as Usual'
- Audit writes can be filtered by user/time/operation etc. as a MongoDB query
- Audit file can be JSON or BSON and goes on the local disk
- Audit logs can be encrypted

Audit best practices

Have separate individual database logins for humans (DBAs)

- Audit everything done ad-hoc by a human
- Audit nothing done by a non-human
- Have service accounts that humans cannot use

Additional Security Features

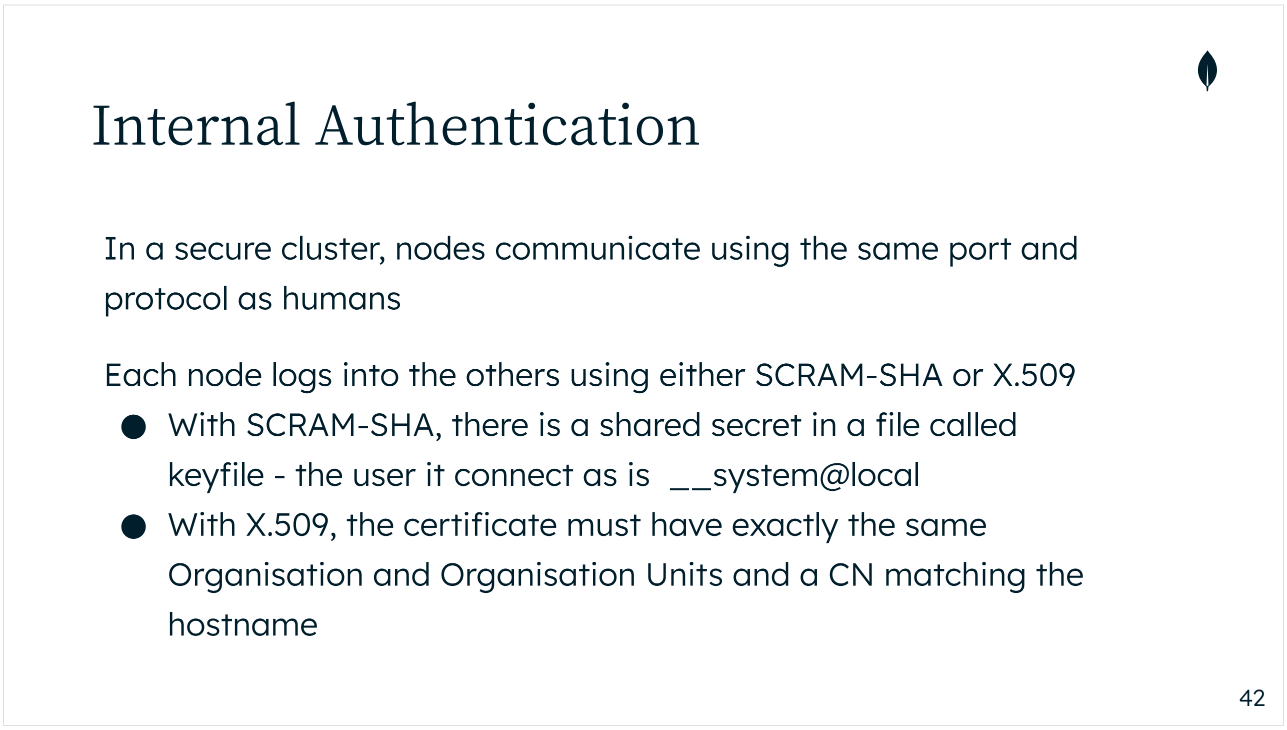
Redacting PII from debug logs is hard - MongoDB has a flag to do it for us at write time

MongoDB only listens on localhost (127.0.0.1) by default

- Access from other IPs must be explicitly granted

Can authorize incoming IPs (CIDRs) for individual users

You can redact entries in MongoDB logs for any PII or other values using the flag redactClientLogData - see https://www.mongodb.com/docs/manual/reference/log-messages/#log-redaction
You can choose what IP Addresses mongodb listens on with the bindIp or bindIpAll config parameters
When creating a user you can configure what addresses they can access from using the authenticationRestrictions clientSource and serverAddress in the user record.



Internal Authentication

In a secure cluster, nodes communicate using the same port and protocol as humans

Each node logs into the others using either SCRAM-SHA or X.509

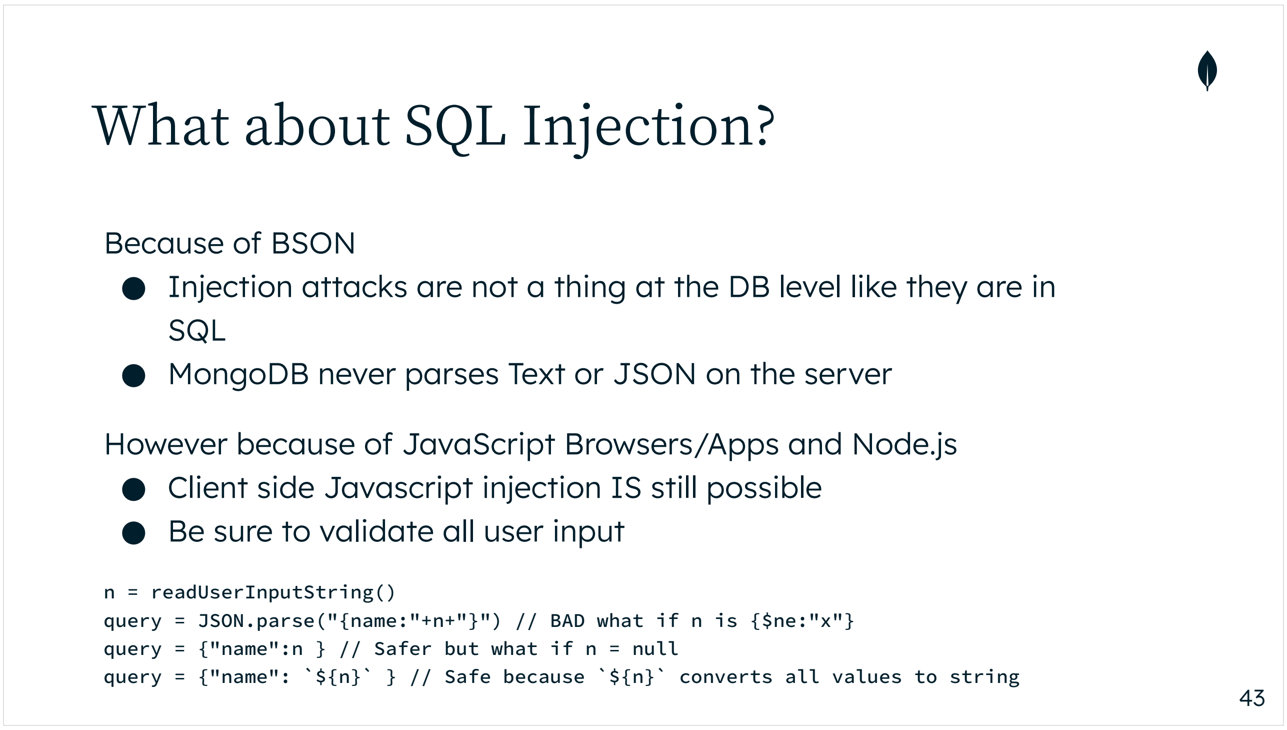
- With SCRAM-SHA, there is a shared secret in a file called keyfile - the user it connect as is __system@local
- With X.509, the certificate must have exactly the same Organisation and Organisation Units and a CN matching the hostname

When we have multiple servers that need to be able to connect to each other and trust each other - they use essentially the same login mechanisms as a human or application logging in.

If we are not using TLS then the login is always done using an internal user called __system@local with their credentials held in a file called the Keyfile supplied to each instance. Keyfile is normally a long random
set of base64 characters.

If we are using TLS and we ensure that each hosts TLS certificate has exactly the same Org and Org units then the hosts can connect and authenticate via X.509 if suitably configured.

If you enable Security in a Replica set or sharded cluster, keyfile internal authentication is enabled by default. System security can be enhanced further by using X.509 internal Authentication.



What about SQL Injection?

Because of BSON

- Injection attacks are not a thing at the DB level like they are in SQL
- MongoDB never parses Text or JSON on the server

However because of JavaScript Browsers/Apps and Node.js

- Client side Javascript injection IS still possible
- Be sure to validate all user input

```
n = readUserInputString()
query = JSON.parse("{name:"+n+"}") // BAD what if n is {$ne:"x"}
query = {"name":n } // Safer but what if n = null
query = {"name": `${n}` } // Safe because `${n}` converts all values to string
```

With SQL bad code combines input with a query to make SQL. This can be used to break security.
input name:

Query = "select * from users where name=${NAME}"
db.execute(Query)

If name is " 'BOB' OR 1 = 1" it will fetch all records. or worse "BOB; DROP TABLE USERS"

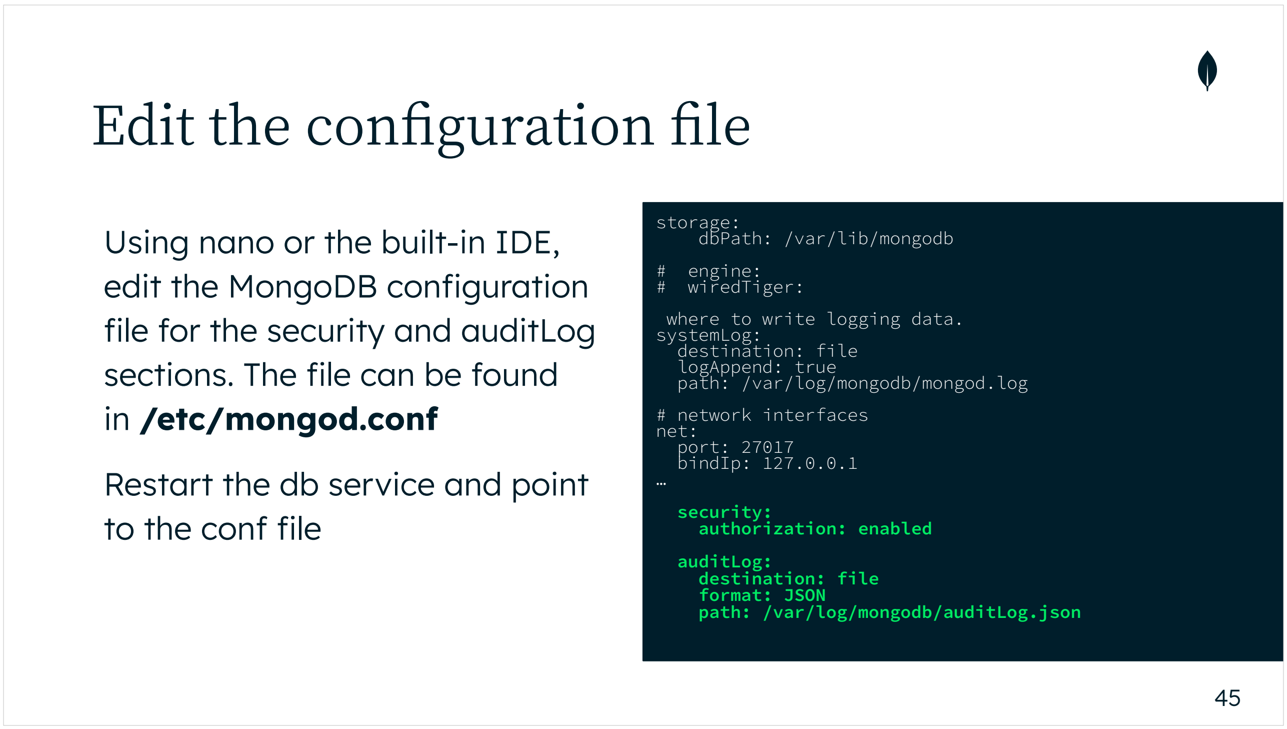
Exercise: Security Access Control

Based on our knowledge so far, we will now learn how to setup access control for a MongoDB database:

1. Edit the configuration file
2. Access the DB and create the first user (what role should this user have?)
3. Create second user with limited access, then with provisional access
4. View the audit logs

IMPORTANT! - this activity is intended to run on Instruqt environments. Run a mongorestore on the dump directory to have sample data for the learners to follow along with on the local mongodb instance.
From the root directory, run the following:
mongorestore ./dump

It is advisable that the first user is userAdminAnyDatabase - with this user, all other users with their corresponding roles can be created.



Edit the configuration file

Using nano or the built-in IDE, edit the MongoDB configuration file for the security and auditLog sections. The file can be found in ***/etc/mongod.conf***

Restart the db service and point to the conf file

```
storage:
    dbPath: /var/lib/mongodb

#  engine:
#  wiredTiger:

 where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

# network interfaces
net:
  port: 27017
  bindIp: 127.0.0.1
…

  security:
    authorization: enabled

  auditLog:
    destination: file
    format: JSON
    path: /var/log/mongodb/auditLog.json
```

```
security:
  authorization: enabled

 auditLog:
  destination: file
  format: JSON
  path: /var/log/mongodb.auditLog.json
```

Shutdown the running instance with `mongod --shutdown --config /etc/mongod.conf`
Restart with `mongod --fork --config /etc/mongod.conf`

Create User Admin

Access the DB using **mongosh**. Localhost exception will apply since there are no created users

Create the first database user with the role of **UserAdminAnyDatabase** in the admin database

Login as the user admin

```
$ mongosh

> use admin

> db.createUser({user: "userAdmin", pwd: passwordPrompt()
, roles: [ "userAdminAnyDatabase" ] })

{ ok: 1 }


> db.auth("userAdmin")

Enter password
******* { ok: 1 }


//verify current user
> db.runCommand({connectionStatus: 1})

{
  authInfo: {
    authenticatedUsers: [ { user: 'userAdmin', db:
'admin' } ],
    authenticatedUserRoles: [ { role:
'userAdminAnyDatabase', db: 'admin' } ]
  },
  ok: 1
}
```

Access the db with mongosh
Switch into the admin db and run the following
//To create the useradmin
db.createUser({user: "userAdmin", pwd: "passwordone", roles: ["userAdminAnyDatabase"] })

//You can authenticate with the new user without exiting the mongosh shell with the following
db.auth("userAdmin")

//Check who the current user is
db.runCommand({connectionStatus: 1})

//list all the roles in the current database
show roles

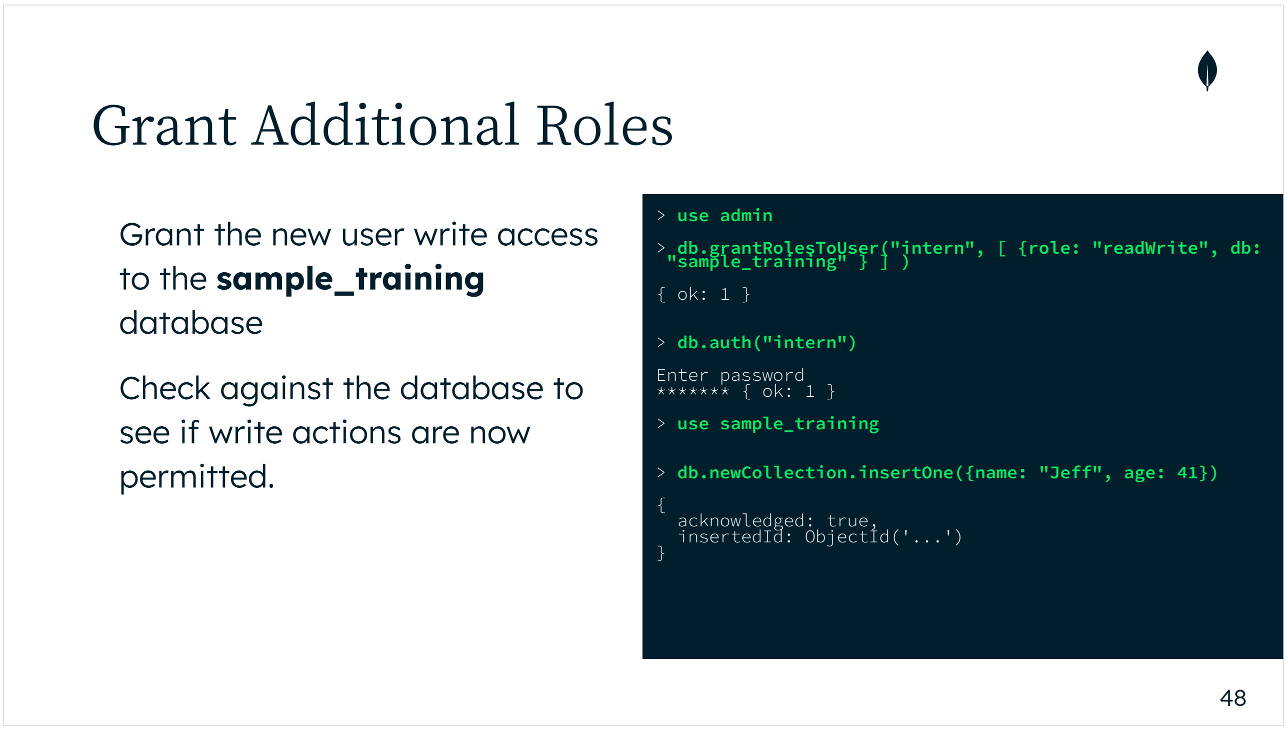
Create Secondary User

Create the second user with basic read privileges on the **sample_training** database.

Authenticate as the new user and confirm they cannot perform write actions.

```
> use admin
> db.createUser({user: "intern", pwd: passwordPrompt(), r
oles: [ {role: "read", db: "sample_training"} ]})
{ ok: 1 }

> db.auth("intern")
Enter password
******* { ok: 1 }
> use sample_training
> db.zips.findOne()
{
  _id: ObjectId(...),
  city: 'ALPINE',
  zip: '35014',
  ...
}
> db.newCollection.insertOne({name: "Jeff", age: 41})
MongoServerError[Unauthorized]: not authorized on
sample_training to execute command ... ✗
```



Grant Additional Roles

Grant the new user write access to the **sample_training** database

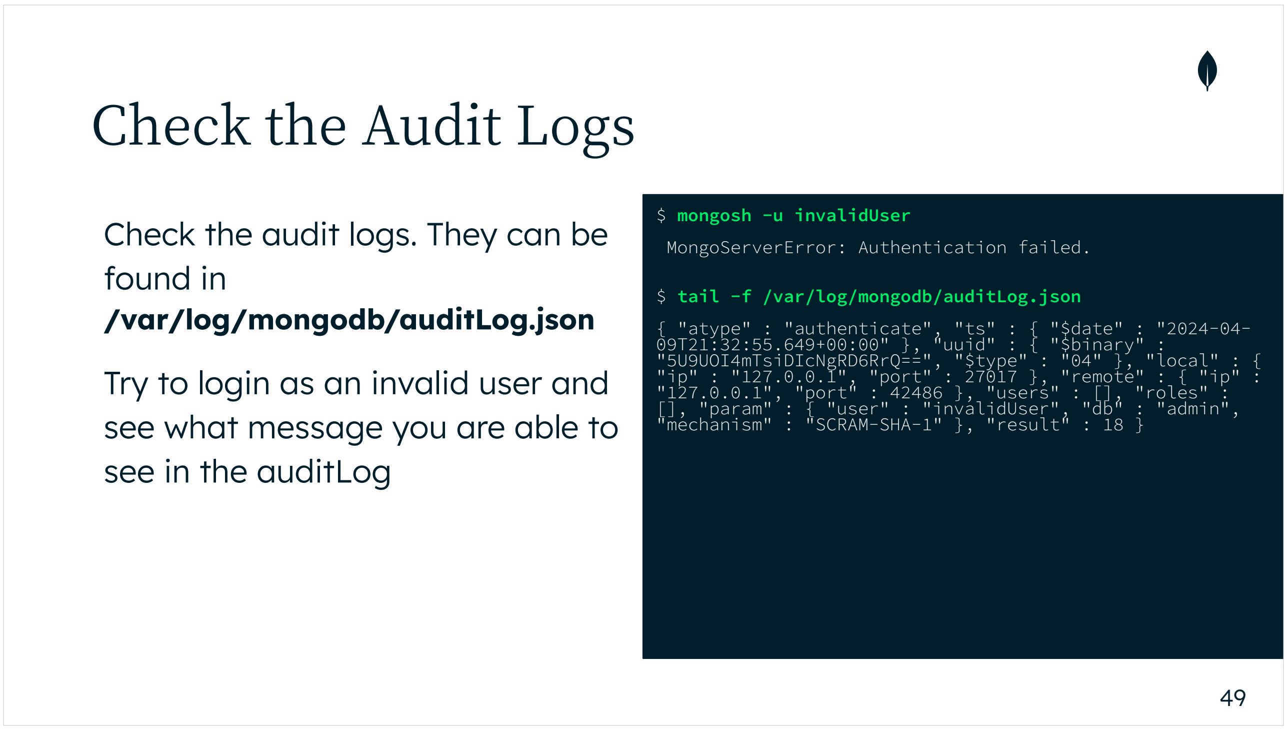
Check against the database to see if write actions are now permitted.

```
> use admin
> db.grantRolesToUser("intern", [ {role: "readWrite", db:
 "sample_training" } ] )
{ ok: 1 }

> db.auth("intern")
Enter password
******* { ok: 1 }
> use sample_training

> db.newCollection.insertOne({name: "Jeff", age: 41})
{
  acknowledged: true,
  insertedId: ObjectId('...')
}
```

//grant additional roles to a user
db.grantRolesToUser("intern", [{role: "readWrite", db: "sample_training" }])



Check the Audit Logs

Check the audit logs. They can be found in **/var/log/mongodb/auditLog.json**

Try to login as an invalid user and see what message you are able to see in the auditLog

```
$ mongosh -u invalidUser
 MongoServerError: Authentication failed.

$ tail -f /var/log/mongodb/auditLog.json
{ "atype" : "authenticate", "ts" : { "$date" : "2024-04-
09T21:32:55.649+00:00" }, "uuid" : { "$binary" :
"5U9UOI4mTsiDIcNgRD6RrQ==", "$type" : "04" }, "local" : {
"ip" : "127.0.0.1", "port" : 27017 }, "remote" : { "ip" :
"127.0.0.1", "port" : 42486 }, "users" : [], "roles" :
[], "param" : { "user" : "invalidUser", "db" : "admin",
"mechanism" : "SCRAM-SHA-1" }, "result" : 18 }
```

Exit the mongosh shell to use the tail command and look at the auditlogs

//tail the auditLog
tail -f /var/log/mongodb/auditLog.json

Quiz Time!

#1. Which of the following are features of MongoDB Security?

A Authentication

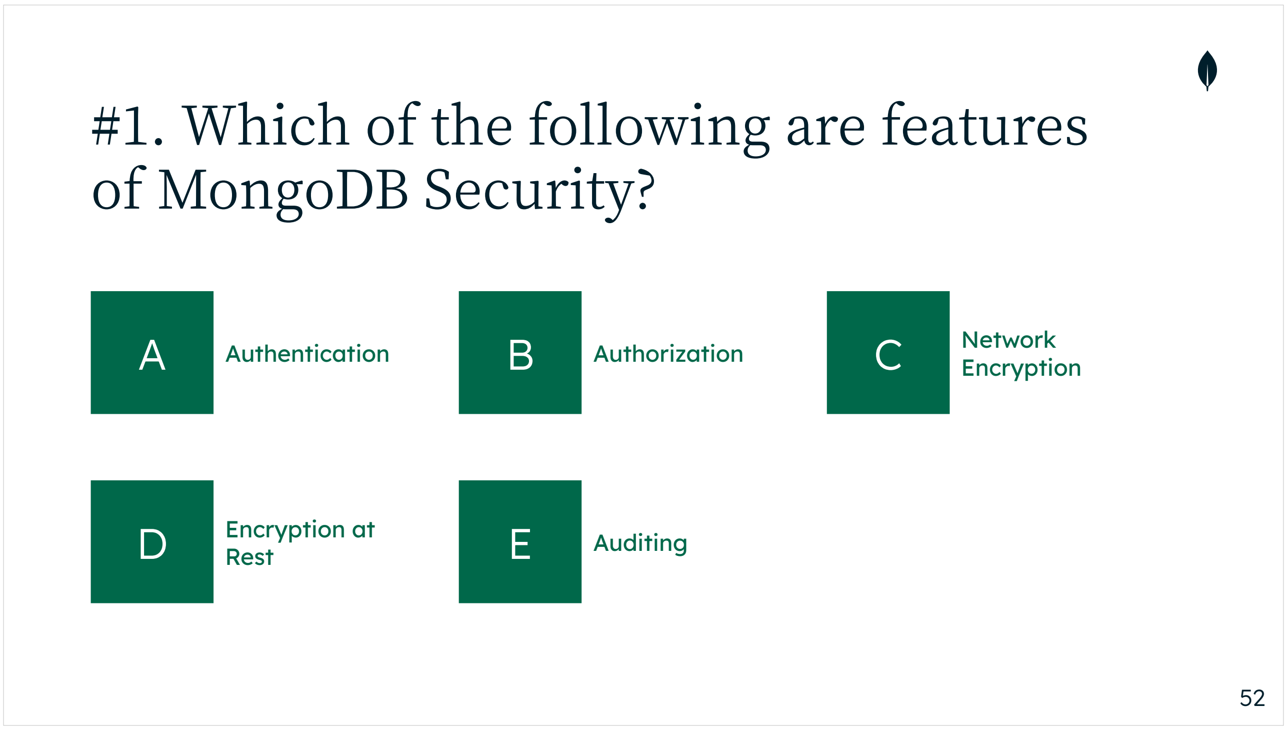
B Authorization

C Network Encryption

D Encryption at Rest

E Auditing

Answer in the next slide.



#1. Which of the following are features of MongoDB Security?

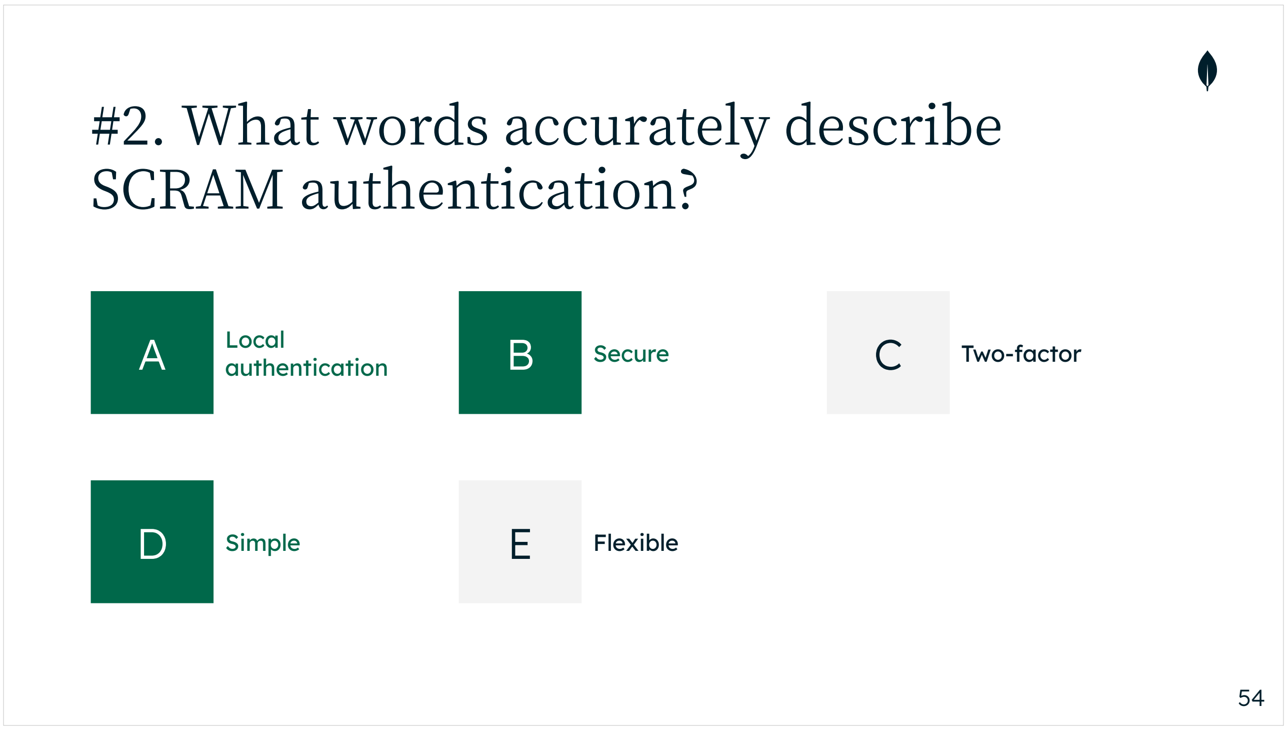
A	Authentication
B	Authorization
C	Network Encryption
D	Encryption at Rest
E	Auditing

All of the above are features of security found in MongoDB.

#2. What words accurately describe SCRAM-SHA256 authentication?

A	Local authentication
B	Secure
C	Two-factor
D	Simple
E	Flexible

Answer in the next slide.



#2. What words accurately describe SCRAM authentication?

[✓ A Local authentication]

[✓ B Secure]

C Two-factor

[✓ D Simple]

E Flexible

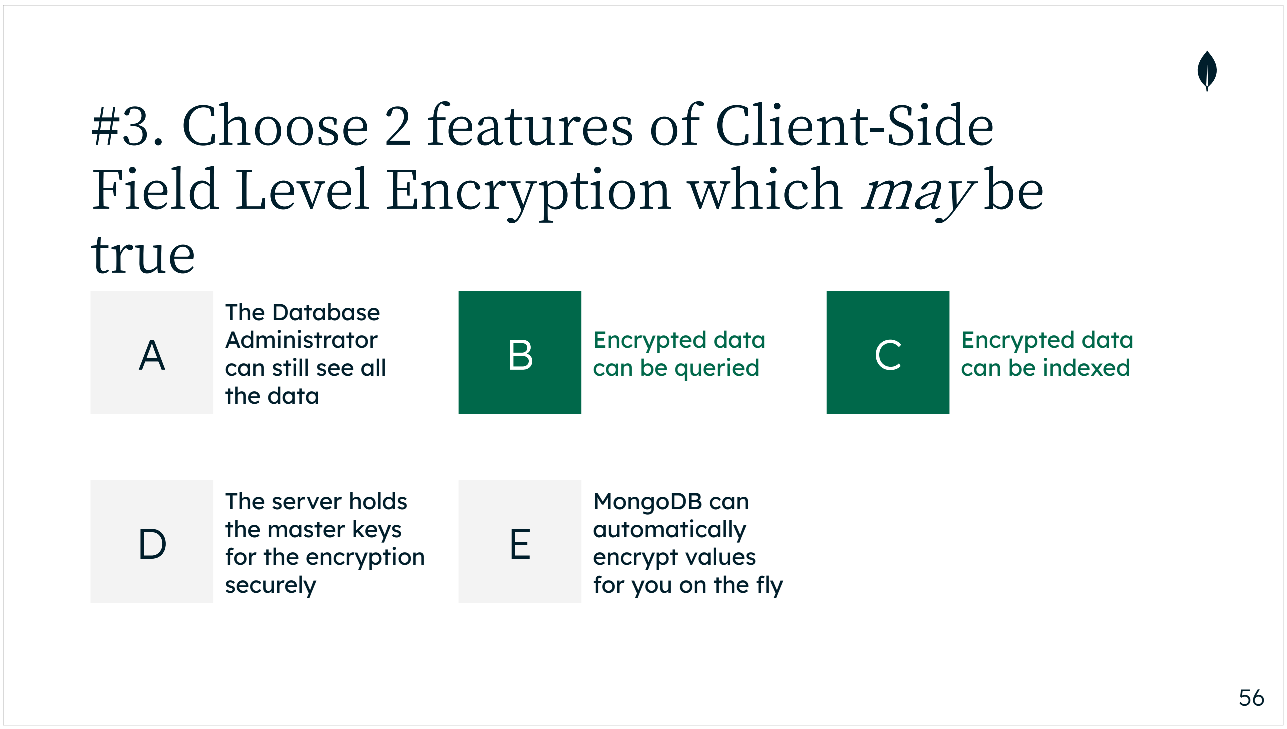
54

SCRAM - Salted Challenge Response Authentication Mechanism. Simple username/password auth secured so that plaintext passwords are never stored. Users must be configured on each DB server instance.

#3. Choose 2 features of Client-Side Field Level Encryption which *may* be true

A	The Database Administrator can still see all the data	B	Encrypted data can be queried	C	Encrypted data can be indexed
D	The server holds the master keys for the encryption securely	E	MongoDB can automatically encrypt values for you on the fly		

Answer in the next slide.



#3. Choose 2 features of Client-Side Field Level Encryption which *may* be true

A — The Database Administrator can still see all the data

[✓ B] — Encrypted data can be queried

[✓ C] — Encrypted data can be indexed

D — The server holds the master keys for the encryption securely

E — MongoDB can automatically encrypt values for you on the fly

56

CSFLE when using deterministic encryption allows you to query on fields that have been encrypted since the value will always show up as the same hash and that still allows of indexing of said field to also still work without losing the referential integrity.

Encrypted field values can be queried and indexed, but only if deterministic encryption was selected.

Recap

Different Authentication methods are appropriate for different types of users

Apply 'Principle of Least Privilege' when assigning roles to users

Data can be encrypted at multiple levels

Secure the physical machines and the network as well as the Database

Always use TLS to encrypt network traffic even in private networks.

Appendix

Encryption

Encryption protects data using a cryptographic key

- Keys allow to lock and unlock things
- Specifically to convert data between readable (plaintext) and unreadable (ciphertext)

Two types of encryption keys:

- Symmetric
- Asymmetric

Symmetric Encryption

Symmetric encryption uses a single key to encrypt and decrypt

- Limited to encryptions and decryption only
- Can be very fast/efficient
- Anyone who has the key can decrypt the message

The more entities that need the key, the greater the risk of compromise

Symmetric encryption is the simplest of the two main types of encryption, and the oldest.

The key is typically a shared secret, but sometimes the method itself is the secret (e.g. Caesar cipher / ROT13).

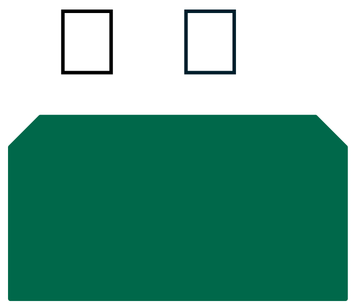
Asymmetric Encryption

Uses pair of encryption keys

- One private
- One public

Imagine a box with two keys

Whichever one is used to lock the box, the other unlocks

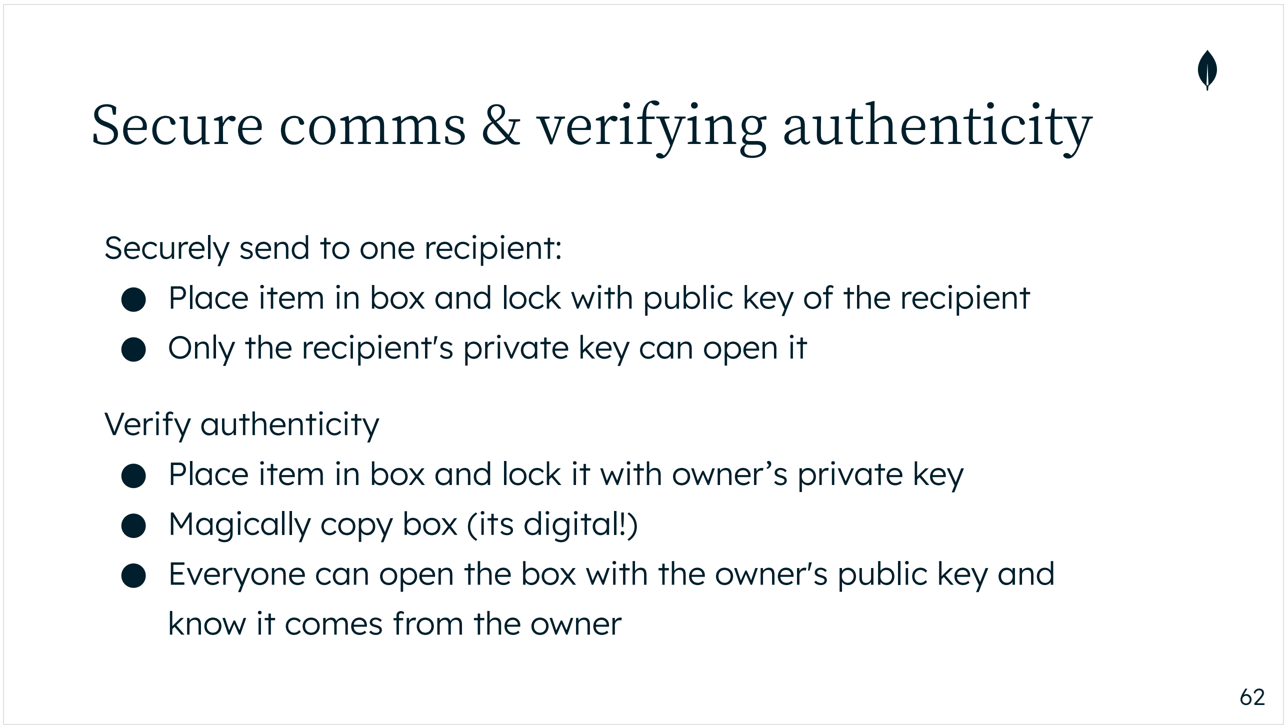


Asymmetric Encryption **(Public-Key Cryptography)** is much slower than Symmetric - but more versatile.

Public-key cryptography uses a key generation protocol (a mathematical function) to generate a key pair. Both the keys are mathematically related to each other in such a way that whichever one is used to encrypt data, the other one can decrypt it.

Think of a box where whichever key is used to lock the box, the other key unlocks the box.

One key in the pair (denoted the private key) is kept secret, the other (public key) is available to anyone who wants it.



Secure comms & verifying authenticity

Securely send to one recipient:

- Place item in box and lock with public key of the recipient
- Only the recipient's private key can open it

Verify authenticity

- Place item in box and lock it with owner's private key
- Magically copy box (its digital!)
- Everyone can open the box with the owner's public key and know it comes from the owner

Using a public-private key pair anyone can send things securely to one person.

Also one person can send to many, and they can be sure who it came from.

Certificates

A Certificate says

- Who we are
- Our public key
- When it's good until
- What we are allowed to do with it
- Who issued it and when

```
   Data:
        Version: 3 (0x2)
        Serial Number:
                
02:37:17:3a:ae:17:19:16:88:3d:0a:0f:0e:b8:7d:e
7
    Signature Algorithm: 
sha256WithRSAEncryption
        Issuer: C=US, O=Amazon, OU=Server CA 
1B, CN=Amazon
        Validity
            Not Before: Jul 10 00:00:00 2020 
GMT
            Not After : Aug 10 12:00:00 2021 
GMT
        Subject: CN=www.mongodb.com
        Subject Public Key Info:
            [... big hex number … ]
            X509v3 Authority Key Identifier: 

            X509v3 Subject Alternative Name: 
                DNS:*.mongodb.com, 
            X509v3 Key Usage: critical
                Digital Signature, Key 
Encipherment
            X509v3 Extended Key Usage: 
                TLS Web Server Authentication
```

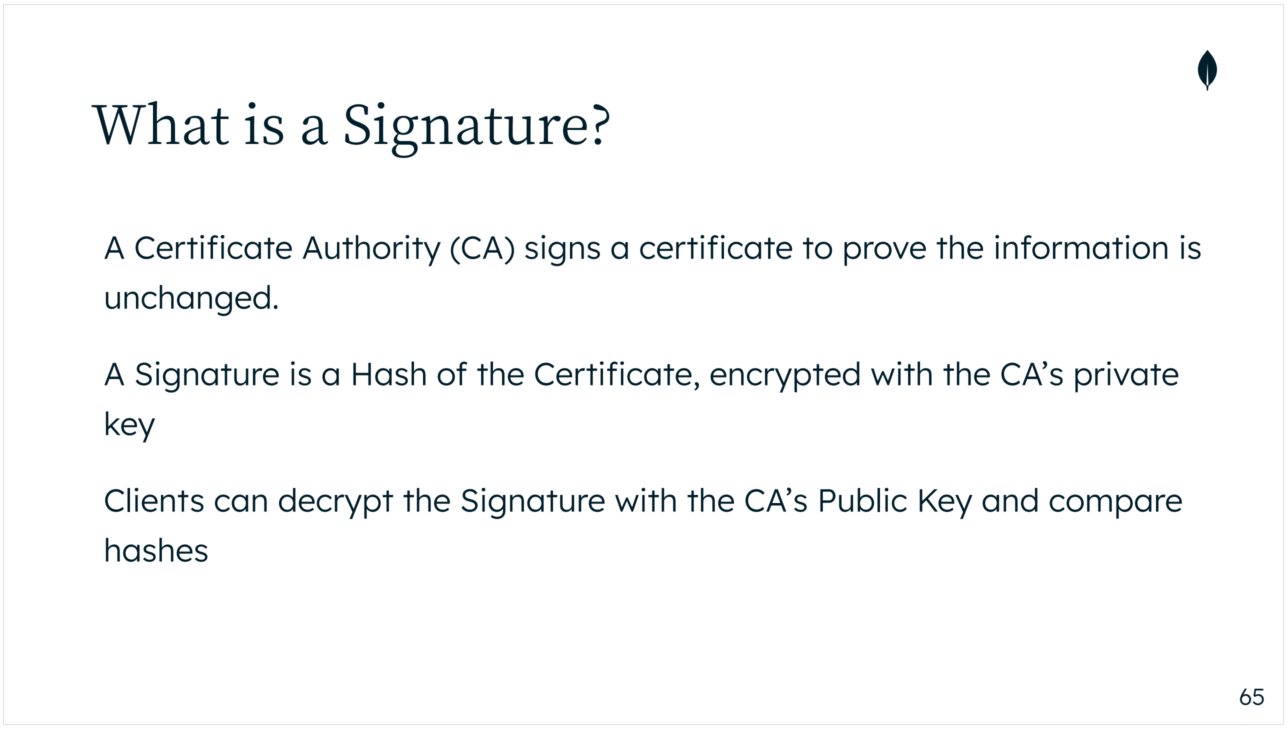
A certificate contains the details about who we are and who the issuer was.
We can then look at the name on the certificate and see if it says they can do something (or are someone)
Starting from MongoDB 4.4 we check whether or not the authority has been revoked.

How can we trust a Certificate?

It's signed - But how do I know the signature isn't fake?

- Is signed by the person who holds it (self-signed)
- Created by someone we trust

How can we trust a certificate is genuine?
Like a paper certificate - anyone can print their own, so we need a way to verify that it was issued by someone qualified to certify them
We also need to know it has not been copied/stolen. Especially on its way to the legal holder.



What is a Signature?

A Certificate Authority (CA) signs a certificate to prove the information is unchanged.

A Signature is a Hash of the Certificate, encrypted with the CA’s private key

Clients can decrypt the Signature with the CA’s Public Key and compare hashes

Certificates can be signed - a signer computes a number (hash) from it then encrypts that hash
If we decrypt the number using the signatories key and it matches what we get when we hash it ourselves
Then it's correctly signed and has not been modified

Acquire the Root CA Public Key: Client should already possess the CA public key (this is the point of trust). The client trusts this root CA to have issued legitimate certificates to legitimate clients. In reality there are intermediate point of trusts (intermediate certificates)
Extract the Signature from the Server's Certificate: The server's digital certificate contains the certificate itself and the digital signature. This signature was created as explained in the step before by the CA by encrypting a hash of the certificate content with the CA's private key. The client extract this digital signature from the digital certificate.
Hash the Certificate Data: The client calculates a hash of the entire certificate data (so repeating the same process the CA did when the certificate was originally signed)
Verify the Signature: The client uses the public key of the root CA to decrypt the digital signature extracted from the server's certificate. This decryption process produces a hash value.
Compare Hashes: The client compares the hash value obtained by decrypting the signature with the hash it computed from the certificate's content. If these two hash values match, it means that the certificate hasn't been tampered with and the signature is valid. If they don't match, the certificate is considered invalid or tampered.

How can we trust a Signature?

The Certificate Authority (CA), which signs the certificate, is known and trusted by us:

- We have their details installed in our OS (Root CA):
  - CA Certificate
  - CA Public Key

OR

- They are trusted by a CA we trust (Intermediate CA)

How does this help trust?

The Certificate Authority (CA) is someone we trust

We have their public key

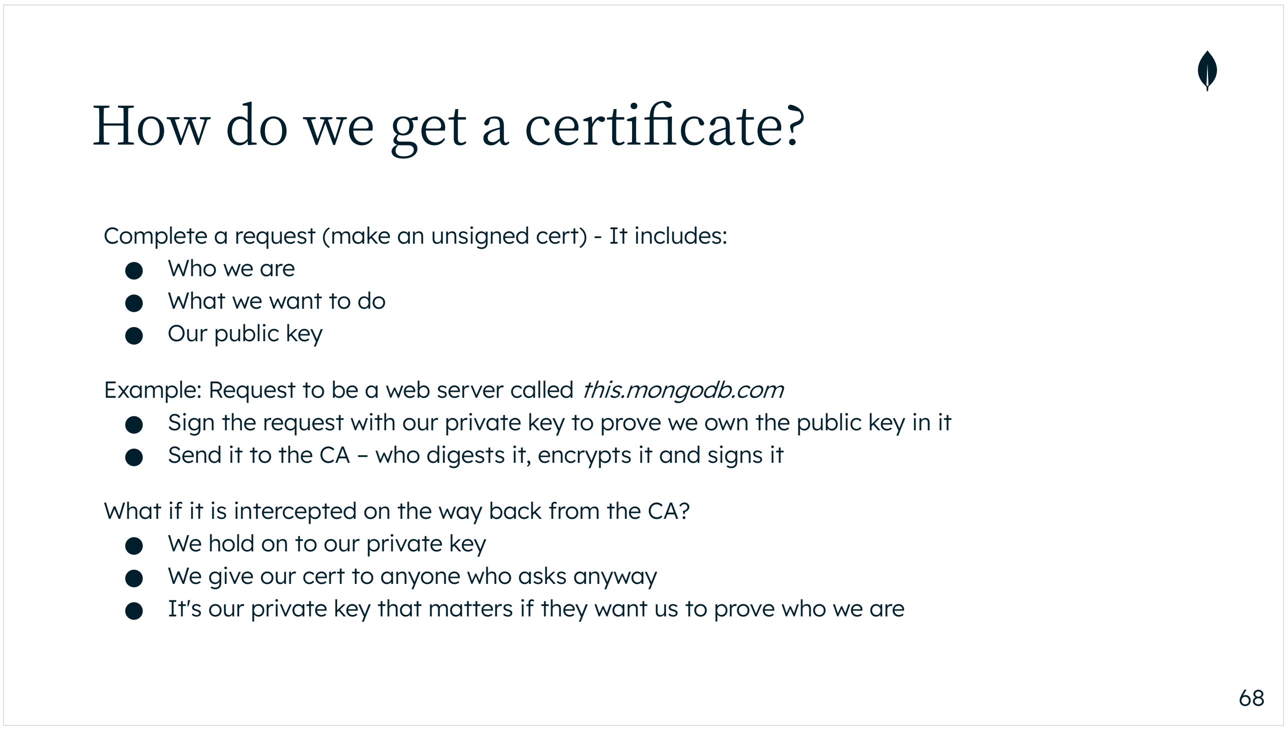
They sign the certificate for the holder

They check the requester is the legitimate owner/holder

There is a “chain of trust” to the CA

We hold the CA Certificate (details and public key)

The role of the Certificate Authority is to act as the root of the chain of trust
They sign the certificates - having first ensured you have access to the domain you are asking for a certificate for
They might email at that domain, or ask you to host a specific file or create a DNS record.
They are often real world companies whose job it is to sign and notarise things



How do we get a certificate?

Complete a request (make an unsigned cert) - It includes:

- Who we are
- What we want to do
- Our public key

Example: Request to be a web server called *this.mongodb.com*

- Sign the request with our private key to prove we own the public key in it
- Send it to the CA – who digests it, encrypts it and signs it

What if it is intercepted on the way back from the CA?

- We hold on to our private key
- We give our cert to anyone who asks anyway
- It's our private key that matters if they want us to prove who we are

How to get a certificate:

- Complete a request using an unsigned certificate
- Send it to the CA
- CA signs and returns

Public Key Infrastructure

Understanding Encryption Keys and Public Key Infrastructure (PKI) is important

It's sometimes not well understood at this level

Don't need to know the maths behind how it works but helps understanding the basics

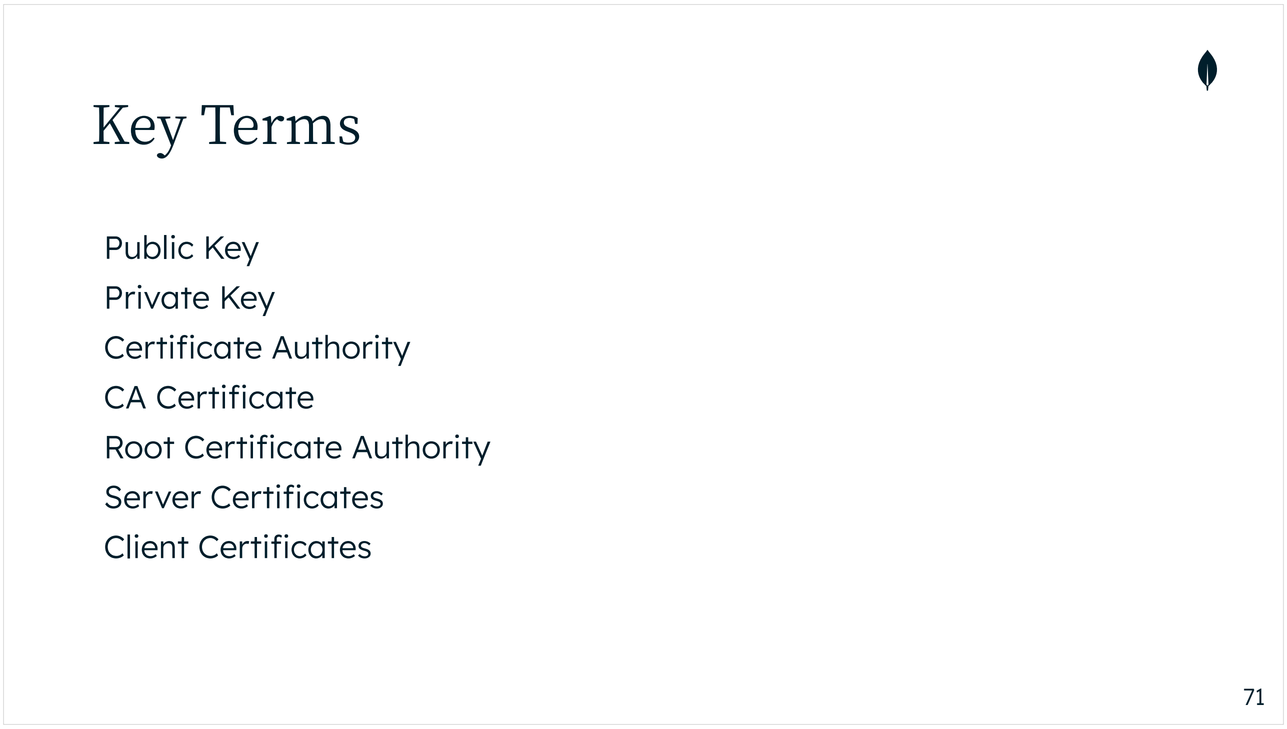
Certificate Authorities

There are top level CAs:
OS and Browsers know their public keys and trust them (Thawte & Verisign)

May issue certificates to be a CA for a specific subdomain – keep checking all way up:

- Host A is signed by CA 1 (who we don't know), but CA 1 uses a certificate that has CA rights and that certificate is signed by CA 2 (who we do know and trust)
- Only need a shared point of trust – can MAKE our own top CA – as long as we give out the CA certificate to anyone who has to trust it

Some companies whole business is being honest broker third parties
If they issue us a certificate - they have checked in the real world we are who we say we are
One permission they can give is permission to issue certificates, perhaps restricted in what for.
When using an intermediate CA to sign certificates we need to include the chain of certificates in our CA bundle when enabling TLS with MongoDB



Key Terms

Public Key
Private Key
Certificate Authority
CA Certificate
Root Certificate Authority
Server Certificates
Client Certificates

Class Exercise