MongoDB Security Architecture

MongoDB

Table of Contents

Introduction

In today's data-rich world database platforms are under constant attack from hackers. The frequency and severity of data breaches continues to escalate. Industry analysts predict cybercrime will cost the global economy $6 trillion annually by 2021.

Organizations face an onslaught of new threat classes and threat actors with phishing, ransomware and intellectual property theft growing more than 50% year on year, and key infrastructure subject to increased disruption. With databases storing an organization's most important information assets, securing them is top of mind for administrators.

This paper discusses best practices with respect to security in MongoDB. First we will look at the requirements for securing modern application data followed by a deeper dive into the features that make up MongoDB's holistic approach to security. While MongoDB Community Server has basic security features like user authentication and authorization, you will find advanced security features like encryption at rest available in MongoDB Enterprise Advanced and the fully-managed MongoDB Atlas global cloud database service.

Like any good airline pilot, if you are looking for a checklist of things to consider for locking down your MongoDB deployment check out the MongoDB Security Checklist. This list provides best practices to follow when self-hosting MongoDB clusters. If you are using MongoDB Atlas, check out the MongoDB Atlas Security Features and Setup for best practices to follow when using the global cloud database service.

MongoDB gives you all the capabilities you need to confidently defend, detect, and control access to valuable and sensitive data, and give you a foundation to meet the mandates of new regulatory compliance initiatives.

Requirements for Securing Modern Application Data

In light of increasing threats over the past decade coupled with heightened concern for individual privacy, industries and governments around the world have embarked on a series of initiatives designed to increase security, reduce fraud and protect personally identifiable information (PII), including:

- PCI DSS for managing cardholder information
- HIPAA standards for managing healthcare information
- GDPR and CCPA for the protection respectively of EU and Californian citizen data privacy
- FISMA to ensure the security of data in the federal government
- FERPA to protect the privacy of student education records
- The Asia Pacific Cross-border Privacy Enforcement Arrangement (CPEA), creating a framework for regional cooperation in the enforcement of privacy laws

In addition to these initiatives, new regulations are being developed every year to cope with emerging threats and new demands for tighter controls governing data use.

Each set of regulations defines security and auditing requirements which are unique to a specific industry or application, and compliance is assessed on a per-project basis.

It is important to recognize that compliance can be achieved only by applying a combination of controls that we can summarize as People, Processes, and Platforms:

- People defines specific roles, responsibilities, and accountability.
- Processes defines operating principles and business practices.
- Platforms defines technologies used for data storage and processing.

Despite differences between different regulations, there are common foundational requirements across all of the directives, including:

- Restricting data access, enforced via predefined privileges and roles
- Measures to protect against accidental or malicious disclosure, loss, destruction, or damage of personal data
- The separation of duties when accessing and processing data
- Recording user, administrative staff, and application activities with a database

These requirements inform the security architecture of MongoDB, with best practices for the implementation of a secure, compliant data management environment.

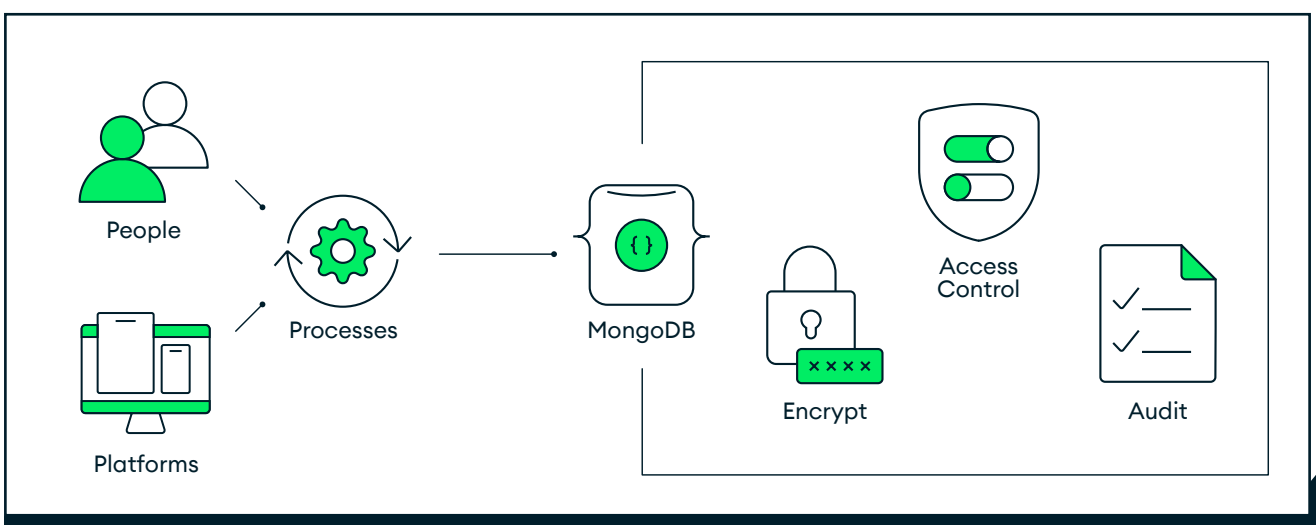


Figure 1: MongoDB End-to-End Security Architecture

A holistic security architecture must cover the following:

- **User access management** to restrict access to sensitive data, implemented through authentication and authorization controls
- **Auditing** database actions for forensic analysis
- **Data protection** via encryption of data in-motion over the network, in-use in the database, and at-rest in persistent storage
- **Environmental protection** to ensure secure host operating system, network, and other database infrastructure

The requirements for each of these elements are as follows:

User Access Management – Authentication

Authentication is designed to confirm the identity of entities accessing the database. In this context, entities are defined as:

- Users who need access to the database as part of their day-to-day business function
- Administrators (i.e. sysadmins, DBAs, QA staff) and developers
- Software systems including application servers, reporting tools, and management and backup suites
- Physical and logical nodes that the database runs on. Databases can be distributed across multiple nodes both for scaling operations and to ensure continuous operation in the event of systems failure or maintenance.

Best practices for authentication are as follows:

Create security credentials

Create login credentials for each entity (user, process or device) that will need access to the database, and avoid creating a single "admin" login that multiple entities share.

By creating credentials it becomes easier to define, manage, and track system access for each entity. Should credentials become compromised, this approach makes it easier to revoke access without disrupting other entities that need to access the database.

Developers, Administrators, and DBAs should all have unique credentials to access the database. When logins are shared it can be impossible to identify who attempted different operations, and it eliminates the ability to assign fine-grained permissions. With unique logins, staff that move off of the project or leave the organization can have their access revoked without affecting other user accounts.

It is a best practice to create separate login credentials for each application that will be accessing the database. As new applications are introduced and old applications retired, this approach helps manage access. Some MongoDB customers create multiple logins for different services within a single application, which are then recorded in audit trails and query logs.

Internal Authentication should be configured between nodes within a replica set and sharded clusters. This prevents unauthorized instances from joining a database cluster, preventing the illicit copying or movement of data to insecure nodes.

Supporting database and centralized user access management

Databases should provide the ability to manage user authentication either within the database itself, typically via a Challenge/Response mechanism, or through integration with organization-wide identity management systems. Integrating MongoDB within the existing information security infrastructure enforces centralized and standardized control over user access. If, for example, a user's access must be revoked, the update can be made in a single repository and enforced instantly across all systems, including MongoDB.

Enforce password policies

Passwords should adhere to minimum complexity requirements established by the organization, and they should be reset periodically. Low entropy passwords can be easy to break, even if they are encrypted. High entropy passwords can be compromised given sufficient time.

User Rights Management – Authorization

Once an entity has been authenticated, authorization governs what that entity is entitled to do in the database. Privileges are assigned to user roles that define a specific set of actions that can be performed against the database. Best practices include:

Grant minimal access to entities

Entities should be provided with the minimal database access they need to perform their function. If an application requires access to a logical database, it should be restricted to operations on that database alone, and prevented from accessing other logical databases. This helps protect against both malicious and accidental access or unauthorized modification of data.

Group common access privileges into roles

Entities can often be grouped into "roles" such as "Developer", "DBA", "Sysadmin", and "App server." Permissions for a role can be centrally managed and users can be added or removed from roles as needed. Using roles helps simplify management of access control by defining a single set of rules that apply to specific classes of entities, rather than having to define them individually for each user.

Control which actions an entity can perform

When granting access to a database, consideration should be made for which specific actions or commands each entity should have permission to run. For example, an application may need read/write permissions to the database, whereas a reporting tool may be restricted to a read-only permission. Some users may be granted privileges that enable them to insert new data to the database, but not to update or delete existing data. Care should be taken to ensure that only the minimal set of privileges is provided. Credentials of the most privileged accounts could compromise the entire database if they are hacked internally or by an external intruder.

Control access to sensitive data

To prevent the emergence of data silos, it should be possible to restrict permissions to individual fields, based on security privileges. For example, some fields of a record may be accessible to all users of the database, while others containing sensitive information, such as PII, should be restricted to users with specific security clearance.

Auditing

By creating audit trails, changes to data and database configuration can be captured for each entity accessing the database, providing a log for compliance and forensic analysis. Auditing can also detect attempts to access unauthorized data.

Track changes to database configuration

Any time a database configuration is changed, the action should be recorded in an audit log which should include the change action, the identity of the entity and a timestamp.

Track changes to data

It should be possible to configure the audit trail to capture every query or write operation to the database. Care, however, should be exercised when configuring this rule for applications. For example, if the application is inserting tens of thousands of records per second, writing each operation to the audit log will impose performance overhead to the database. The project team should determine any tradeoffs between performance and auditing requirements. It should be possible to filter events that are captured, for example only specific users, IP addresses or operations.

Encryption

Encryption is the encoding of critical data whenever it is in transit, at rest, or in-use, enabling only authorized entities to read it. Data will be protected in the event that eavesdroppers or hackers gain access to the server, network, filesystem or database.

Encrypt connections to the database

All user or application access to the database should be via encrypted channels including connections established through the drivers, command line or shell, as well as remote access sessions to the database servers themselves. Internal communications between database nodes should also be encrypted, i.e. traffic replicated between nodes of a database cluster.

Encrypt data at rest

One of the most common threats to security comes from attacks that bypass the database itself and target the underlying Operating System and physical storage of test/QA and production servers or backup devices, in order to access raw data. On-disk encryption of the database's data files and backups mitigates this threat.

Sign and rotate encryption keys

Encryption keys for network and disk encryption should be periodically rotated. TLS encryption channels should use signed certificates to ensure that clients can certify the credentials they receive from server components.

Enforce strong encryption

The database should support FIPS (Federal Information Processing Standard) 140-2 to ensure the implementation of secure encryption algorithms.

Encrypt data in-use

When your requirements demand the highest level of data protection, leverage client-side field-level encryption (FLE). FLE performs the encryption and decryption on the client, ensuring data is never exposed in plaintext while on the server. With FLE, even a comprised administrator account reading memory dumps can not decrypt your data.

Environmental and Process Control

The environment in which the database and underlying infrastructure is running should be protected with both physical and logical controls. These are enforced in the underlying deployment environment, rather than in the database itself, and include:

- Installation of firewalls
- Network configurations
- Defining file system permissions
- Creation of physical access controls to the IT environment

As configuration errors and unpatched systems are one of the largest causes of attackers bypassing security mechanisms, there are a series of operational processes that should be adopted to further promote and enforce secure operation, including:

- DBA and developer training
- Database provisioning, monitoring and backup
- Database maintenance, i.e. applying the latest patches

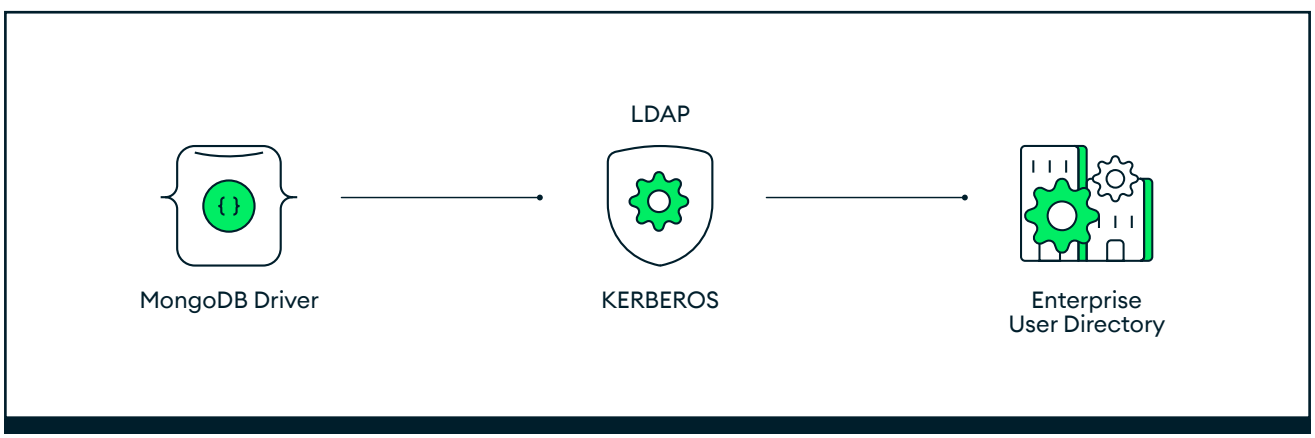


Figure 2: Integrating MongoDB with Centralized User Access Controls

MongoDB Security Features

With comprehensive controls for user rights management, auditing and encryption, coupled with best practices in environmental protection, MongoDB can meet the best practice requirements described earlier.

The following section discusses MongoDB's security architecture. Also refer to the MongoDB Security Checklist for a list of security measures that you should implement to protect your MongoDB installation.

MongoDB Enterprise Advanced

MongoDB Enterprise Advanced is the certified and supported production release of MongoDB for running on your infrastructure, with advanced security features,including LDAP authentication and authorization, Kerberossupport, encryption of data at-rest, FIPScompliance, and maintenance of audit logs.

These capabilities extend MongoDB's already comprehensive security framework,which includes Role-Based Access Control, PKI certificates, TLS data transport encryption, Read-Only views and Client-side Field Level Encryption. You can get access to all of the features of MongoDB Enterprise free of charge for evaluation and development environments.

While MongoDB Enterprise Advanced is the focus of this paper, the key capabilities of MongoDB Atlas are summarized in the "Why MongoDB Atlas?" section of this paper as occasional references are made to Atlas throughout this paper. MongoDB Atlas is a global cloud database service available on all of the major cloud platforms. Download the MongoDB Atlas Security Controls whitepaper to learn more about the specific security architecture of the Atlas service.

Enable Access Control and Enforce Authentication

Beginning with version 3.6 and above, MongoDB only allows connections originating from the local host. By default, all remote connections are denied unless the matching IP address or CIDR range was explicitly added to the MongoDB configuration. This concept of off-by-default and IP whitelisting is also used within MongoDB Atlas.

Before you bind to external IP addresses, you should enable access control and evaluate other security measures listed in Security Checklist to prevent unauthorized access.

MongoDB Authentication

Authentication is the process of validating the identity of the entity making a connection to MongoDB. MongoDB supports a number of authentication mechanisms including:

- SCRAM (Default)
- x.509 Certificate Authentication
- LDAP proxy authentication
- Kerberos authentication

These mechanisms allow MongoDB to integrate into your existing authentication system and meet the requirements of different environments.

Database authentication

When adding an entity such as a user, you create the user in a specific database. This database is known as the “authentication database” for the user. A user can have privileges across different databases; that is, a user’s privileges are not limited to their authentication database.

If your application architecture allows, it is easier from a management standpoint to use a specific database like “admin” rather than creating the entities within their respective application databases. Outside of the management benefit, there is no other advantage with either strategy. If you choose to create users in databases other than “admin”, make sure to include the database component identifying the authentication database in the connection string.

```
use admin
db .createUser(
{
user: “reportsUser”,
pwd: “12345678”,
roles: [
{ role: “read”, db: “reporting” },
{ role: “read”, db: “products” },
{ role: “read”, db: “sales” },
{ role: “readWrite”, db: “accounts” }
]
}
)
```

LDAP authentication

LDAP is widely used by many organizations to standardize and simplify the way large numbers of users are managed across internal systems and applications. In many cases, LDAP is also used as the centralized authority for user access control to ensure that internal security policies are compliant with corporate and regulatory guidelines. With LDAP integration, MongoDB Enterprise Advanced can both authenticate and authorize users directly against existing LDAP infrastructure to leverage centralized access control architectures.

Review the LDAP integration documentation to learn more about LDAP and MongoDB Enterprise Advanced.

Kerberos authentication

With MongoDB Enterprise Advanced, authentication using a Kerberos service is supported. Kerberos is an industry standard authentication protocol for large client/server systems, allowing both the client and server to verify each others’ identity. With Kerberos support, MongoDB can take advantage of existing authentication infrastructure and processes, including Microsoft Active Directory.

Before users can authenticate to MongoDB using Kerberos, they must first be created and granted privileges within MongoDB.

The process for doing this, along with a full configuration checklist is described in the MongoDB and Kerberos tutorial.

x.509 certificate authentication

With support for x.509 certificates MongoDB can be integrated with existing information security infrastructure and certificate authorities, supporting both user and inter-node authentication.

Users can be authenticated to MongoDB using client certificates rather than self-maintained passwords.

Inter-cluster authentication and communication between MongoDB nodes can be secured with x.509 member certificates rather than key files, ensuring stricter membership controls with less administrative overhead, i.e. by eliminating the shared password used by key files. x.509 certificates can be used by nodes to verify their membership of MongoDB replica sets and sharded clusters.

MongoDB User Defined Roles Permit Separations of Duty	
Application	Application Server Role: • Read and Write on Application Database
Reporting	Bi Role: • Read only on Application Database
ETL	
DBA	DBA Role: • Read and Write on Application Database • Administration on Application Database • Administration on MongoDB cluster

You can leverage a single or separate Certificate Authorities (CAs) for certificate validation on a server instance based on the communication type.

For example, use one CA between inbound or outbound client and server communications, and another CA for internal cluster communications between cluster members.

Regardless if you are using certificates or key files, both of these can be modified without having to take down your cluster even when modifying the Distinguished Name.

Instructions for configuration are described in the MongoDB and x.509 certificates tutorial.

MongoDB and Microsoft Active Directory

MongoDB Enterprise Advanced provides support for authentication using Microsoft Active Directory with both Kerberos and LDAP. The Active Directory domain controller authenticates the MongoDB users and servers running in a Windows network, again to leverage centralised access control.

MongoDB Authorization

Authorization is the process of determining the specific permissions an entity has on the database. MongoDB employs Role-Based Access Control (RBAC) to govern access to a MongoDB system. A user is granted one or more roles that determine the user's access to database resources and operations. Outside of role assignments, the user has no access to the system.

Role-based access control

MongoDB provides predefined roles out of the box supporting common user and administrator database privileges such as dbAdmin, dbOwner, clusterAdmin, readWrite and many more.

These roles can be further customised through User Defined Roles, enabling administrators to assign fine-grained privileges to clients, based on their respective data access and processing needs. To simplify account provisioning and maintenance, roles can be delegated across teams, ensuring the enforcement of consistent policies across specific data processing functions within the organization. MongoDB provides the ability to specify user privileges with both database and collection-level granularity.

Privileges are assigned to roles, and roles are in turn assigned to users.

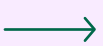
For example:

- Classes of users and applications can be assigned privileges to insert data, but not to update or delete data from the database
- DBAs may be assigned privileges that enable them to create collections and indexes on the database, while developers are restricted to CRUD operations
- Certain administrator roles may have cluster-wide privileges to build replica sets and configure sharding, 7 while others are restricted to creating new users or inspecting logs
- Processes for monitoring MongoDB clusters can be restricted to run just those commands that retrieve server status, without having full administrative access to perform database operations
- Within a multi-tenant environment, "landlord" developers and administrators can be assigned permissions across physical databases, while "tenant" developers and administrators can be granted a more limited set of actions across logical databases or individual collections. This functionality enables a clear separation of duties and control, both between and within organizations.

When combined with the auditing capabilities available with MongoDB Enterprise Advanced, customers can define specific administrative actions per role, and then log all of those actions.

As a result, the organization is able to enforce end-to-end operational control and maintain insight of actions for compliance and reporting. We discuss auditing in more detail later in this guide.

Review the Authorization section of the documentation to learn more about roles in MongoDB.



LDAP authorization

In addition to LDAP authentication, MongoDB Enterprise Advanced also support authorization via LDAP. This enables existing user privileges stored in the LDAP server to be mapped to MongoDB roles, without users having to be recreated in MongoDB itself. When configured with an LDAP server for authorization, MongoDB will allow user authentication via LDAP, Active Directory, Kerberos, or X.509 without requiring local user documents in the $external database.

Note: The $external database is used when users have credentials stored externally to MongoDB. When a user successfully authenticates, MongoDB will perform a query against the LDAP server to retrieve all groups the LDAP user is a member of, and will transform those groups into their equivalent MongoDB roles. LDAP authentication and authorization can be configured either via the command line, or for additional administrative convenience, via the Ops Manager GUI.

Field-level security using read-only views

To enforce field-level security, developers and DBAs have multiple approaches available to them. We discuss Client-Side Field Level Encryption later, while this section focuses on using Read Only Views to enforce field-level security.

You can define non-materialized views that expose only a subset of data from an underlying MongoDB collection, i.e. a view that filters out specific fields, such as Personally Identifiable Information (PII) from sales data or health records. As a result, risks of data exposure are dramatically reduced. DBAs can define a view of a collection that's generated from an aggregation over another collection(s) or view. Permissions granted against the view are specified separately from permissions granted to the underlying collection(s). This capability allows organizations to more easily meet compliance standards in regulated industries by restricting access to sensitive data, without creating the silos that emerge when data has to be broken apart to reflect different access privileges.

Views can also contain computed fields – for example summarizing total and average order value per region, without exposing underlying customer data. All of this can be done without impacting the structure or content of the original source collections. Developers and DBAs can modify the underlying collection's schema without impacting applications using the view.

Views are defined using the standard MongoDB Query Language and aggregation pipeline. They allow the inclusion or exclusion of fields, masking of field values, filtering, schema transformation, grouping, sorting, limiting, and joining of data using $lookup and $graph Lookup to another collection.

Learn more about read-only views from the MongoDB documentation.

Log redaction

MongoDB Enterprise Advanced can also be configured with log redaction to prevent potentially sensitive information, such as personal identifiers, from being written to the database's diagnostic log.

Developers and DBAs who may need to access the logs for database performance optimization or maintenance tasks still get visibility to metadata, such as error or operation codes, line numbers, and source file names, but are unable to see any personal data associated with database events.

MongoDB Auditing

The MongoDB Enterprise Advanced auditing framework logs all access and actions executed against the database. The auditing framework captures administrative actions (DDL) such as schema operations as well as authentication and authorization activities, along with read and write (DML) operations to the database.

Administrators can construct and filter audit trails for any operation against MongoDB, whether DML, DCL or DDL without having to rely on third party tools. For example, it is possible to log and audit the identities of users who accessed specific documents, and any changes they made to the database during their session.

Administrators can configure MongoDB to log all actions or apply filters to capture only specific events, users or roles. The audit log can be written to multiple destinations in a variety of formats including to the console and syslog (in JSON format), and to a file (JSON or BSON), which can then be loaded to MongoDB and analyzed to identify relevant events.

Each MongoDB server writes events to its attached storage. The DBA can then use their own tools to merge these events into a single audit log, enabling a cluster-wide view of operations that affected multiple nodes.

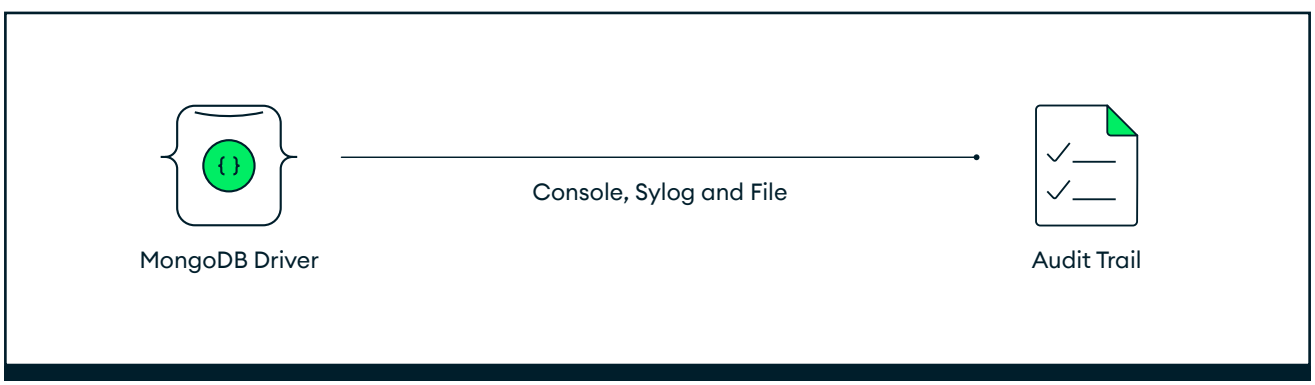


Figure 3: MongoDB Maintains an Audit Trail of Administrative Actions Against the Database

MongoDB Enterprise Advanced also supports role-based auditing. It is possible to log and report activities by specific role, such as user Admin or dbAdmin – coupled with any inherited roles each user has – rather than having to extract activity for each individual administrator.

Auditing adds performance overhead to a MongoDB system. The amount is dependent on several factors including which events are logged and where the audit log is maintained, such as on an external storage device and the audit log format. Users should consider the specific needs of their application for auditing and their performance goals in order to determine their optimal configuration.

Learn more from the MongoDB auditing documentation.

MongoDB Encryption

Administrators can encrypt MongoDB data in transit over the network and at rest in permanent storage and backups. Users can encrypt data at the field level protecting sensitive information from administrators and other legitimate users while data is in use on the server.

Network encryption

Support for TLS allows clients to connect to MongoDB over an encrypted channel. Clients are defined as any entity capable of connecting to the MongoDB server, including:

- Users and administrators
- Applications
- MongoDB tools (e.g., mongodump, mongorestore, mongotop)
- Nodes that make up a MongoDB cluster, such as replica set members, query routers and config servers.

It is possible to mix encrypted with non-encrypted connections on the same port, which can be useful when applying finer grained encryption controls for internal and external traffic, as well as avoiding downtime when upgrading a MongoDB cluster to support TLS.

The TLS protocol is also supported with x.509 certificates and supports Forward Secrecy, providing the assurance that your users' session keys will remain secure even if the server's private key is compromise.

MongoDB Enterprise Advanced supports FIPS 140-2 encryption if run in FIPS Mode with a FIPS validated Cryptographic module. The mongod and mongos 9 processes should be configured with the "sslFIPSMode" setting. In addition, these processes should be deployed on systems with an OpenSSL library configured with the FIPS 140-2 module.

The MongoDB documentation includes a tutorial for configuring TLS connections.

Data at rest encryption

There are multiple ways to encrypt data at rest with MongoDB. Encryption can be implemented at the application level, or via external filesystem and disk encryption solutions. By introducing additional technology into the stack, both of these approaches can add cost, performance overhead and operational complexity.

With the MongoDB Encrypted storage engine, protection of data at-rest becomes an integral feature of the database. By natively encrypting database files on disk, administrators eliminate both the management and performance overhead of external encryption mechanisms. The Encrypted storage engine provides an additional level of defense, allowing only those staff with the appropriate database credentials access to encrypted data.

Using the Encrypted storage engine, the raw database content, referred to as plaintext, is encrypted using an algorithm that takes a random encryption key as input and generates ciphertext that can only be read if decrypted with the decryption key. The process is entirely transparent to the application. MongoDB supports a variety of encryption schema, with AES-256 (256 bit encryption) in CBC mode being the default. AES-256 in GCM mode is also supported. The encryption schema can be configured for FIPS 140-2 compliance.

The storage engine encrypts each database with a separate key. The keywrapping scheme in MongoDB wraps all of the individual internal database keys with one external master key for each server.

The Encrypted storage engine supports two key management options – in both cases, the only key being managed outside of MongoDB is the master key:

- Local key management via a keyfile.
- Integration with a third party key management appliance via the KMIP protocol (recommended).

Most regulatory requirements mandate that the encryption keys must be rotated and replaced with a new key at least once annually.

MongoDB can achieve key rotation without incurring downtime by performing rolling restarts of the replica set. When using a KMIP appliance, the database files themselves do not need to be re-encrypted, thereby avoiding the significant performance overhead imposed by key rotation in other databases. Only the master key is rotated, and the internal database keystore is re-encrypted.

The Encrypted storage engine is designed for operational efficiency and performance:

- Compatible with WiredTiger's document level concurrency control and compression.
- Support for Intel's AES-NI equipped CPUs for acceleration of the encryption/ decryption process.
- As documents are modified, only updated storage blocks need to be encrypted, rather than the entire database.

Based on user testing, the Encrypted storage engine minimizes performance overhead to around 10-15% (this can vary, based on data types being encrypted), which can be much less than the observed overhead imposed by some filesystem encryption solutions.

The Encrypted storage engine is based on WiredTiger and available as part of MongoDB Enterprise Advanced.

Refer to the documentation to learn more, and see a tutorial on how to configure the storage engine.

Client-side field level encryption

Whenever the database server handles the encryption and decryption of data, regardless of the database vendor, users with elevated privileges such as administrators can potentially read memory leveraged by the operating system process hosting the database. To completely protect your data from a compromised administrator account or a system administrator the database engine should never expose the plain text of the encrypted data at any point in the query lifecycle.

With MongoDB's Client-side Field Level Encryption (FLE), you can selectively encrypt individual document fields, each optionally secured with its own key. Our implementation of Field Level Encryption is totally separated from the database, making it transparent to the server, and instead handled exclusively within the MongoDB drivers on the client. All encrypted fields on the server – stored in-memory, in system logs, at-rest, and in backups – are rendered as ciphertext, making them unreadable to any party who does not have client access along with the keys necessary to decrypt the data. This is a different and more comprehensive approach than column encryption used in many relational databases. As most handle encryption server-side, data is still accessible to administrators who have access to the database instance itself, even if they have no client access privileges.

In developing Client-side Field Level Encryption, MongoDB has worked with two of the world's leading authorities on database cryptography, including a co-author of the IETF Network Working Group Draft on Authenticated AES encryption. Drawn from academia and industry, these teams have provided expert guidance on our FLE design and reviewed the FLE software implementation.

Field Level Encryption serves as an important addition to your defense-in-depth security strategy. Consider a typical MongoDB deployment from a risk management perspective:

- With filesystem encryption alone, system administrators or attackers who elevate system-level user privileges still have access to plaintext database files on both server storage and in memory.
- The MongoDB Encrypted Storage Engine provides a way to mitigate filesystem and backup file access risks by encrypting all MongoDB data before it is written to disk, and ensuring keys are non-persistent. Any attacker obtaining database files from the filesystem would be unable to read them. However, administrators and compromised authenticated database users still have access to the underlying data on a running instance.
- Using MongoDB FLE you can now protect individual fields with all key management, encryption, and decryption operations occurring exclusively outside the database server. With FLE enabled, a compromised administrator or user obtaining access to the database, the underlying filesystem, or the contents of server memory (for example via scraping or process inspection) will only see unreadable encrypted data. While storage engine encryption and FLE can be used independently, they bring the greatest levels of protection when used together.

Advantages of MongoDB Client-side Field Level Encryption include:

- **Automatic, transparent encryption:** Application code can run unmodified for most database read and write operations when FLE is enabled. Other client-side approaches require developers to modify their query code to use the explicit encryption functions and methods in a language SDK.
- **Separation of duties:** System administrators who traditionally have access to operating systems, the database server, logs, and backups cannot read encrypted data unless explicitly given client access along with the keys necessary to decrypt the data.

- **Regulatory Compliance:** Comply with "right to be forgotten" conditions in new privacy regulations such as the GDPR – simply destroy the customer key and the associated personal data is rendered useless.
- **Minimal performance penalty:** As encryption is handled on the client, impacts to server performance are minimal when working with encrypted fields.

For isolation, each field can be encrypted with its own unique key natively integrated with external key management services backed by FIPS 140-2 validated Hardware Security Modules (HSMs) such as Amazon's KMS.

To understand more about the implementation of FLE, let's take a look at the flow of a query submitted by an authenticated client, represented in the following figure:

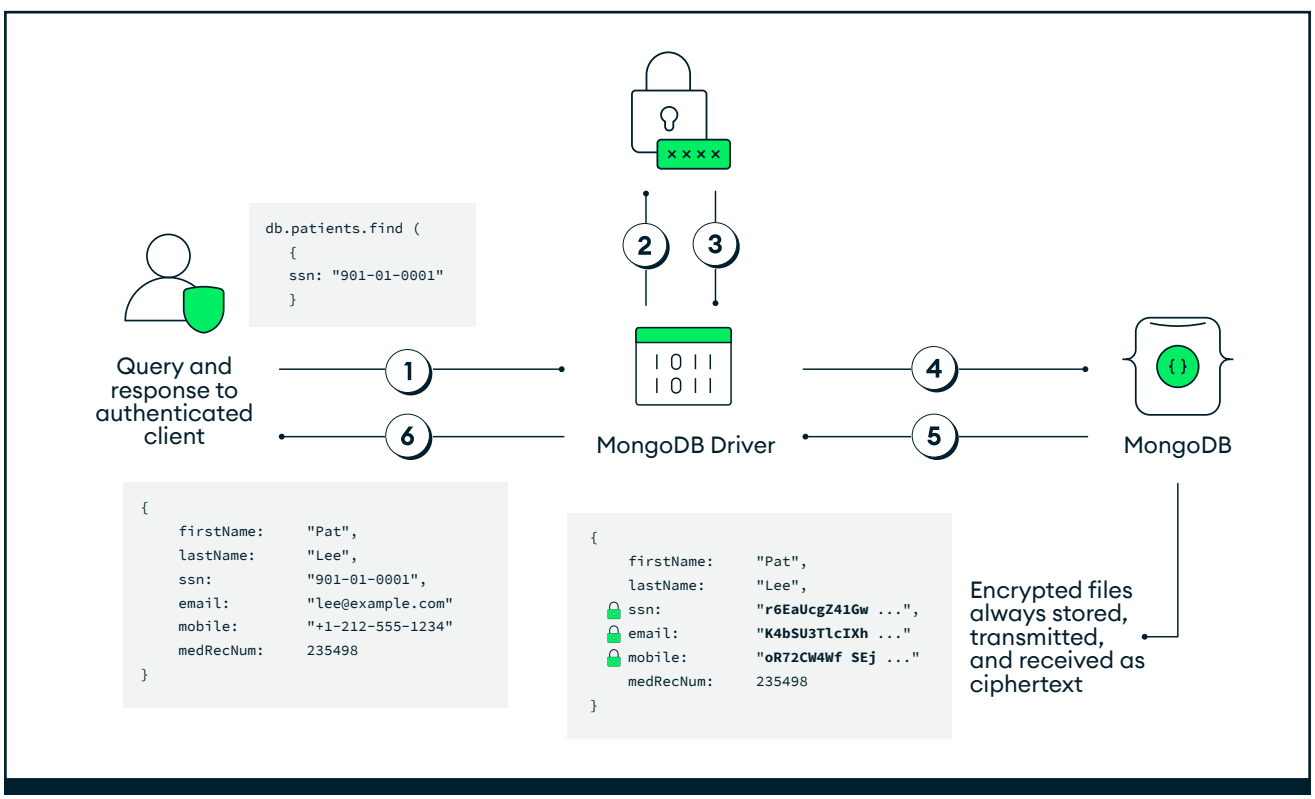


Figure 4: Client-side Field Level Encryption data flow

Upon receiving the query, the MongoDB driver uses per-field encryption rules specified within a JSON schema validation to determine if any encrypted fields are involved in the filter. The driver requests the fields' encryption keys from the external key manager.

The key manager returns the keys to the driver, which then encrypts the sensitive fields. The driver submits the query to the MongoDB server with the encrypted fields rendered as ciphertext. MongoDB returns the encrypted results of the query to the driver. The query results are decrypted by the keys held by the driver, and returned to the client.

Since the database server has no access to the encryption keys, certain query operations such as sorts and range based queries on encrypted fields are not possible unless implementing a client side solution, which may have additional performance impacts.

As a result, FLE is best applied to selectively protect just those fields containing highly sensitive, personally identifiable data such as credit card information and social security numbers.

Client-side Field Level Encryption is especially powerful when using managed database services like MongoDB Atlas. In Atlas, all cluster storage and backups are encrypted at rest by default. You can provide an additional layer of encryption by protecting the database keys using the cloud providers' key management service.

With the addition of FLE, keys are protected in an isolated, customermanaged AWS KMS account (additional key management solutions are coming online in the near future). Atlas Site Reliability Engineers and Product Engineers have no mechanism to access FLE KMS keys, rendering data unreadable to any MongoDB personnel managing the underlying database or infrastructure for you.

By combining these security capabilities, you eliminate common security concerns when moving database workloads to managed services in the cloud. This is because you both control and manage the encryption keys, rather than have the database operator manage the keys for you.

For more information see the MongoDB Server 4.2 release notes and documentation and Client-side Field Level Encryption in the MongoDB Documentation.

Environment & Processes

Building on the database security controls discussed above, running MongoDB in a trusted environment, implementing a secure development lifecycle, and enforcing deployment best practices further reduces the risk of security breaches.

A "Defense in Depth" approach is recommended to complement secure MongoDB deployments, addressing a number of different methods for managing risk and reducing exposure.

The intention of a Defense in Depth approach is to layer your environment to ensure there are no exploitable single points of failure that could allow an intruder or untrusted party to access the data stored in the MongoDB database.

Secure environments use the following strategies to control access, with more detail available in the Network Exposure and Security section of the documentation.

Network Filter: By using filters such as firewalls and router ACL rules, connections to MongoDB from unknown systems can be blocked. Firewalls should limit both incoming and outgoing traffic to/from a specific port to trusted and untrusted systems. For best results and to minimize overall exposure, ensure that only traffic from trusted sources can reach mongod and mongos instances and that the mongod and mongos instances can only connect to trusted outputs. In addition, unneeded system services should be deactivated.

Binding IP Addresses: The bind_ip setting for mongod and mongos instances limits the network interfaces on which MongoDB programs will listen for incoming connections.

Running in VPNs: Limit MongoDB programs to non-public local networks and virtual private networks. Virtual Private Networks (VPNs) make it possible to link two networks over an encrypted and limited-access trusted network. Typically MongoDB users configure SSL rather than IPSEC protocols for performance advantages.

Dedicated OS User Account: A user account dedicated to MongoDB should be created and used to run MongoDB executables. MongoDB should not run as the "root" user.

File System Permissions: The servers running MongoDB should employ filesystem permissions that prevent users from accessing the data files created by MongoDB. MongoDB configuration files and the cluster keyfile should be protected to disallow access by unauthorized users.

Query Injection: As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks should not pose a risk to the system for queries submitted as BSON objects.

However, several MongoDB operations permit the evaluation of arbitrary JavaScript expressions and care should be taken to avoid malicious expressions. Fortunately, most queries can be expressed in BSON and for cases where Javascript is required, it is possible to mix JavaScript and BSON so that user-specified values are evaluated as values and not as code.

MongoDB also allows the administrator to configure the MongoDB server to prevent the execution of Javascript scripts.

This will prevent MapReduce jobs from running, but the aggregation pipeline can be used as an alternative in many use cases.

Physical Access Controls: In addition to the logical controls discussed above, controlling physical access to servers, storage and backup media provides critical environmental protection.

Leverage secret managers: Storing passwords in configuration files complicates auditing and increases risk for system compromise.

Database monitoring & upgrading

Proactive monitoring of all components within an IT environment is always a best practice. System performance and availability depend on the timely detection and resolution of potential issues before they present problems to users.

From the perspective of database security, monitoring is critical to identifying potential exploits in real time, thereby reducing the impact of any breach. For example, sudden peaks in the CPU and memory loads of host systems and high operations counters in the database can indicate a Denial of Service attack. MongoDB ships with a variety of tools including mongostat and mongotop that can be used to monitor your database.

The most comprehensive monitoring solution is provided by MongoDB Ops Manager, which is the simplest way to run MongoDB on your own infrastructure. Ops Manager makes it easy for operations teams to monitor, secure, back up, and scale MongoDB. Ops Manager is available with MongoDB Enterprise Advanced. MongoDB Cloud Manager is a hosted management tool for MongoDB providing many of the same capabilities as Ops Manager.

Figure 5: Ops Manager Offers Charts, Custom Dashboards & Automated Alerting

MongoDB Cloud Manager is a hosted management tool for MongoDB providing many of the same capabilities as Ops Manager.

Featuring charts, custom dashboards, and automated alerting, Ops and Cloud Manager track 100+ key database and systems health metrics including operations counters, memory and CPU utilization, replication status, open connections, queues and any node status. Cloud Manager can also alert you if any host is Internet-exposed.

The metrics are securely reported to Ops Manager where they are processed, aggregated, alerted and visualized in a browser, letting administrators easily determine the health of MongoDB in real time. Views can be based on explicit permissions, so project team visibility can be restricted to their own applications, while systems administrators can monitor all MongoDB deployments across the organization.

Ops Manager allows administrators to set custom alerts when key metrics are out of range. Alerts can be sent via SMS and email or integrated into existing incident management systems such as PagerDuty, Slack, HipChat and others to proactively warn of potential issues before they escalate to costly outages.

Ops Manager also enables administrators to roll out upgrades and patches to the database without application downtime. By using either the GUI or API, updated packages can be pushed and applied to each server through a series of rolling restarts, all without operator intervention.

Disaster recovery: backups & point-in-time recovery

Data can be compromised by a number of unforeseen events: failure of the database or its underlying infrastructure, user error, malicious activity, or application bugs.

With a backup and recovery strategy in place, administrators can restore business operations by quickly recovering their data, enabling the organization to meetregulatory and compliance obligations.

Ops Manager backups are maintained continuously, just a few seconds behind the operational system. If MongoDB experiences a failure, the most recent backup is only moments behind, minimizing exposure to data loss. Ops Manager and Cloud Manager offers point-in-time recovery of replica sets and cluster-wide snapshots of sharded clusters. You can restore to precisely the moment you need, quickly and safely, allowing users to quickly recover from attacks that corrupt the underlying data.

Queryable backups allow you to connect to backup snapshots and issue read-only queries. This alleviates the burden of performing a full database restore in order to obtain records that may have been accidentally deleted.

Training & consulting services

MongoDB provides extensive training and consulting services to help customers apply best security practices:

- The MongoDB Security course is a no-cost, 3-week online training program delivered by MongoDB University.
- MongoDB University also offers a range of both public and private training for developers and operations teams, covering best practices in using and administering MongoDB.
- MongoDB Global Consulting Services offer a range of packages covering Health Checks, Production Readiness Assessments, and access to Dedicated Consulting Engineers. The MongoDB consulting engineers work directly with your teams to guide development and operations, ensuring skills transfer to your staff.

Keep up to date

Always ensure you are running the latest production-certified release of both MongoDB and the drivers, and have applied the latest set of patches. While MongoDB Enterprise Advanced customers get access to emergency patches, fixes for security vulnerabilities are available to all MongoDB users as soon as they are released.

MongoDB Atlas: Database as a Service For MongoDB

MongoDB Atlas is a cloud database service that makes it easy to deploy, operate, and scale MongoDB in the cloud by automating time-consuming administration tasks such as database setup, security implementation, scaling, patching, and more.

MongoDB Atlas is available on-demand through a pay-as-you-go model and billed on an hourly basis.

It's easy to get started – use a simple GUI to select the public cloud provider, region, instance size, and features you need. MongoDB Atlas provides:

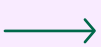
- Security features to protect your data, with fine-grained access control and end-to-end encryption
- Built in replication for always-on availability.
- Geographically dispersed database that can provide lowlatency writes from anywhere in the world. Data can also be easily replicated and geographically distributed, enabling multi-region fault tolerance and fast, responsive reads.
- Fully managed, continuous and consistent backups with point in time recovery to protect against data corruption, and the ability to query backups in-place without full restores
- Fine-grained monitoring and customizable alerts for comprehensive performance visibility
- One-click scale up, out, or down on demand. MongoDB Atlas can provision additional storage capacity as needed without manual intervention.
- Automated patching and single-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features
- Live migration to move your self-managed MongoDB clusters into the Atlas service with minimal downtime

MongoDB Atlas can be used for everything from a quick Proof of Concept, to test/QA environments, to powering production applications.

The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that disruption is minimal if you decide to manage MongoDB yourself and migrate to your own infrastructure.

Built and run by the same team that engineers the database, MongoDB Atlas is the best way to run MongoDB in the cloud.

Learn more or deploy a free cluster now.



Conclusion

With databases storing an organization's most important information assets, securing them is an essential first step in countering new threat classes and actors.

As demonstrated in this white paper, with MongoDB Enterprise Advanced organizations benefit from extensive capabilities to defend, detect and control access to valuable and sensitive data.

You can get started by reviewing the MongoDB Security Documentation or downloading the MongoDB Atlas Security Controls whitepaper to learn more about the specific security architecture of the Atlas service.

Evaluate MongoDB Enterprise Advanced. Download today.