DS130

Client-Side Field Level Encryption

CSFLE Workshop

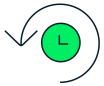
Overview

Learning Objectives

At the end of this lesson, learners will be able to:

- Describe common use cases for CSFLE
- Describe the main features of MongoDB CSFLE
- Describe the key components of MongoDB CSFLE
-

At a Glance



Length:
2 Days

Level:
Advanced

Prerequisites:
DF100
DF200

Agenda

Why use CSFLE

CSFLE Features

CSFLE Components and Details

Explicit Encryption and Decryption

Implicit Encryption and Decryption

Use Cases

Key Rotation

Our Environment

Our Environment

We have three VMs in each student's environment:

csfle-1

- hostname: csfle-mongodb-**PETNAME**.mdbtraining.net
- Has a single-member replica set running that use TLS and authn/authz

csfle-2

- hostname: csfle-kmip-**PETNAME**.mdbtraining.net
- Has a KMIP device running on the VM within a Docker container. This uses TLS for data protection and authentication

csfle-3

- hostname: csfle-client-**PETNAME**.mdbtraining.net
- This is your client. It has Python3 and Pymongo installed, plus programs for Golang and Java
- The VM also has mongosh installed
- The home directory for ec2-user has Python, Golang and Java skeleton code for all exercises

Strigo Tour

You environment will take about 10 minutes to fully boot and be operational.

Skeleton code is available in the following folder:

```
/home/ec2-user/csfle_skeleton_workshop
```

Code is optimised for readability, not performance!

Go users will need to execute the following to find the Go command:

```
source ~/.bash_profile
```

See the appendices for further information on developing with Go

Why use CSFLE?

Sensitive Data needs additional protection

Protect fields

Secure data independently from the Database

Separate Access

Require access to data to be explicitly granted

Maintain Compliance

Comply with regulatory requirements for data privacy

CSFLE Features

MongoDB Client-Side Field Level Encryption Features

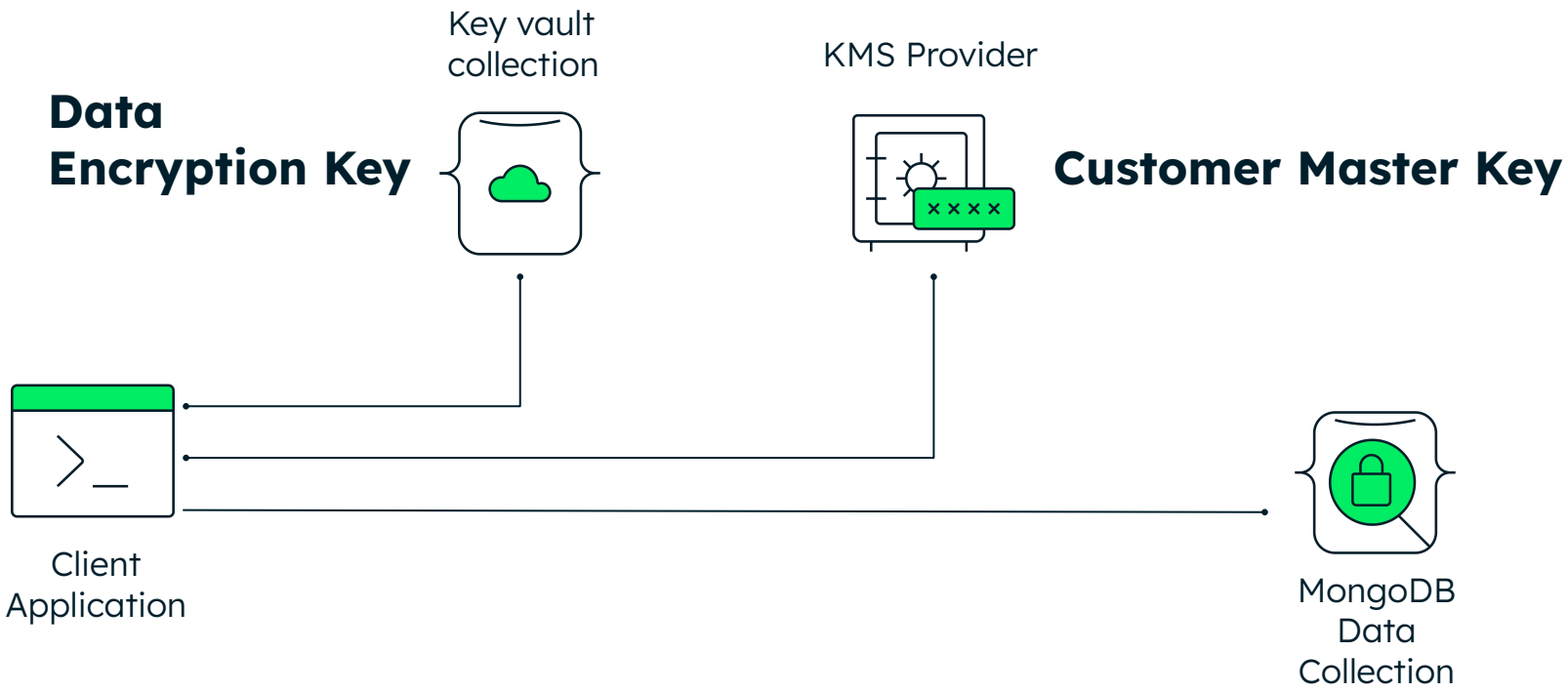
- Encryption and decryption performed at the driver, not the server
- Explicit (manual) or Automatic field encryption
- Deterministic or random encryption algorithm modes
- Enforced field encryption via Schema Validation
- Keys can be per field, document, user, collection, database, or hybrid
- Most BSON types can be encrypted
  - Including arrays and embedded documents

CSFLE Components

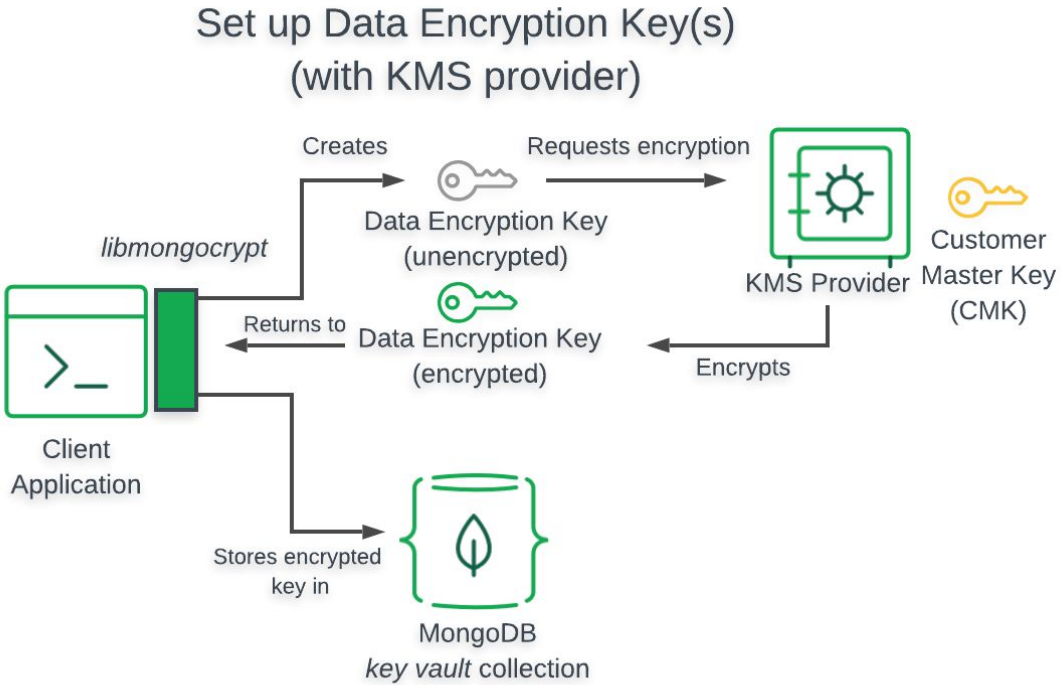
CSFLE Components

- Keys
  - Customer Master Key (CMK)
  - Data Encryption Key (DEK)
- Key Management System (KMS) / Key Provider
- Key Vault
- libmongocrypt
- mongocryptd / crypt_shared
- Encryption Schema

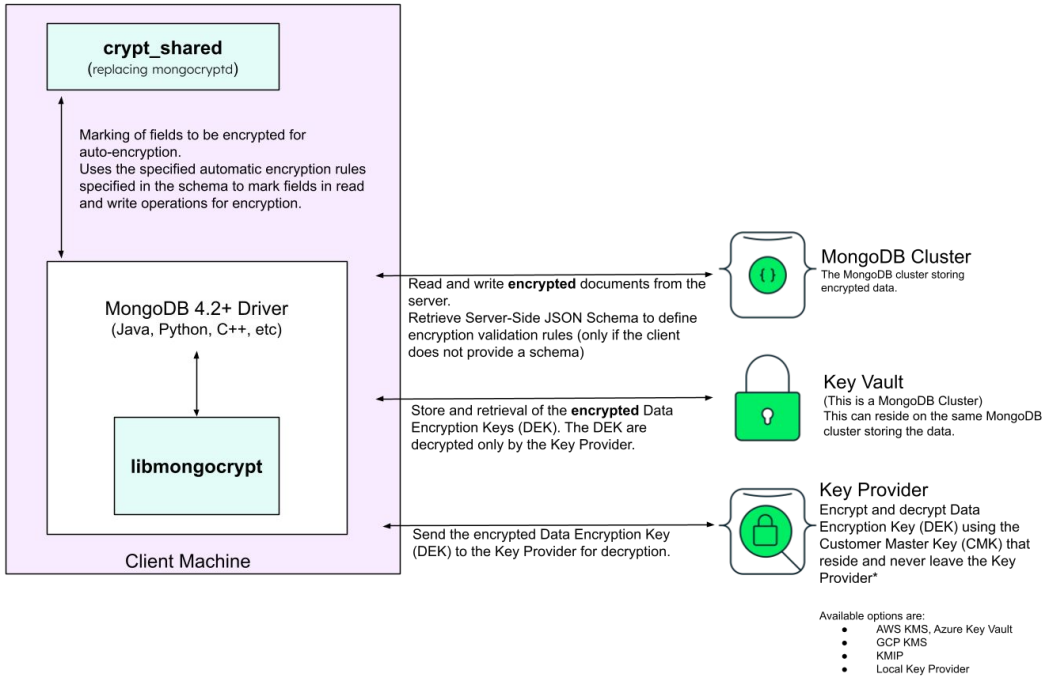
CSFLE Components - Keys



CSFLE Details

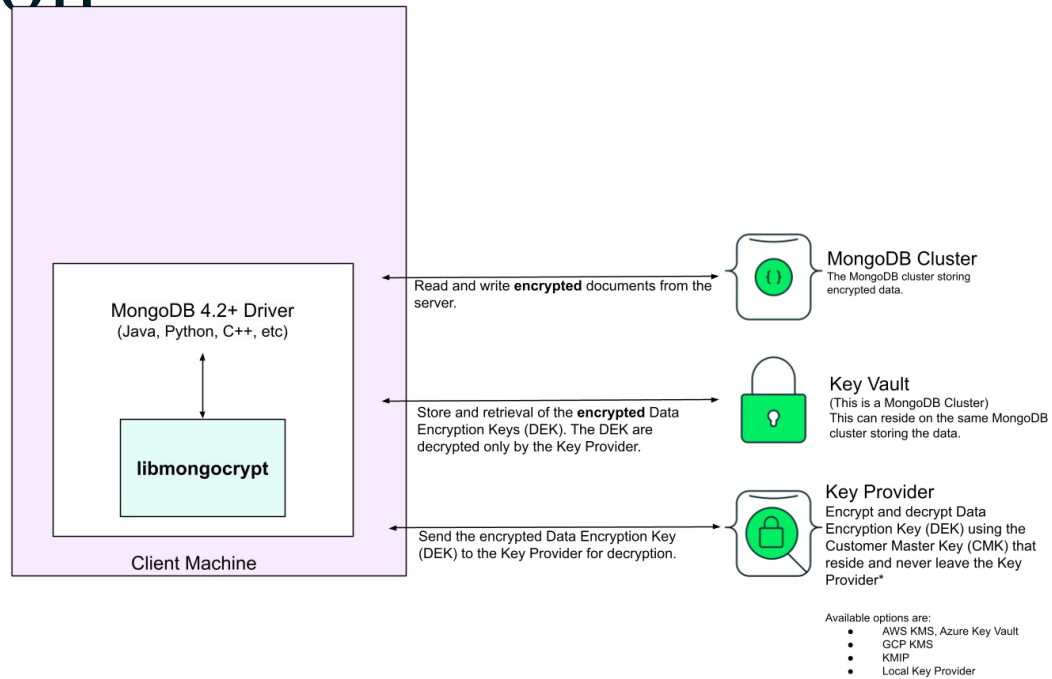


CSFLE Components - Automatic Encryption



* If using KMIP, the CMK leave the KMIP and is cached unencrypted for 1 minute by the libmongocrypt.
Using a Local Key Provider is for third party devices or software where MongoDB does not have an in-built provider

CSFLE Components - Manual Encryption



* If using KMIP, the CMK leave the KMIP and is cached unencrypted for 1 minute by the libmongocrypt.
Using a Local Key Provider is for third party devices or software where MongoDB does not have an in-built provider

Encrypted Data Encryption Key

_id: UUID of DEK

keyMaterial: Encrypted DEK

masterKey: Key Provider and CMK which encrypted the DEK

```
{
 _id:
UUID("c47855db-8ea5-4fe0-afdf-bfc09ecac474"),
 keyAltNames: [ 'dataKey1' ],
 keyMaterial: Binary(Buffer.from("01020....",
"hex"), 0), creationDate:
ISODate("2022-07-20T06:19:28.502Z"),
 updateDate: ISODate("2022-07-20T06:19:28.502Z"),
 status: 0,
 masterKey: {    provider: 'aws',
   region: 'ap-southeast-2',
   key: '6a822d91-15b0-4d8f-b1bb-2f180b9e1d2a'
 }
}
```

CSFLE Components - KMS

- Key Management Systems (KMS) / Key Providers can be either remote or local
- KMSs store the Customer Master Key (CMK)

Local

- Stored on local filesystem
- Insecure

Remote

- Secure offsite storage
- Access auditing
- Key Lifecycle management

Always use a remote KMS for production

CSFLE Components - Key Providers

Most of the key providers are fairly self explanatory, but the Local KMS provider needs further clarification. In the MongoDB documentation the examples often use the Local key provider and create an on-disk file containing the key: this is not a recommended practice. The Local key provider can be used if the customer has a KMS that is not listed and they have a safe and secure method to retrieve the CMK and store it in memory (e.g. never on disk).

For this training course we are going to use the KMIP key provider. We have created a CMK already for you.

Exercise: View CMK ID in KMS

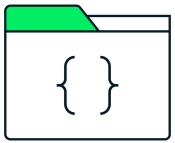
On your client (csfle-3) run the following command at the command line to see the keys in the KMIP key provider:

```
python3 ~/kmip/show_cmks.py
```

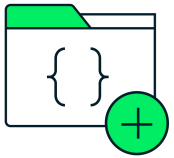
Note the the Unique Identifier value - this is the ID of your CMK:

```
$ python3 ~/kmip/show_cmks.py
('2', [Attribute(attribute_name=AttributeName(value='Unique Identifier'),
attribute_index=None, attribute_value=UniqueIdentifier(value='2')),
...
```

CSFLE Components - Key Vault



MongoDB Collection containing encrypted DEK Documents



Created in advance

Unique index on **keyAltNames** field

Client app should have read-only permissions

Create Collection with Index - Mongosh

To create the collection we would execute the following as a privileged user, which actually creates the collection intrinsically when creating the index:

```
db.getSiblingDB("__encryption").getCollection("__keyVault").createIndex(
  {
    keyAltNames: 1
  },
  {
    unique: true,
    partialFilterExpression: {
      "keyAltNames": {
        "$exists": true
      }
    }
  }
)
```

Exercise: Create Key Vault Collection

On your **csfle-3** Strigo instance, use mongosh to create the Key Vault collection.

Your root user credentials are:
Username: mongoadmin
Password: passwordone

`createIndex()` automatically creates the Database and Collection if they do not already exist

```
$ mongosh
"mongodb://mongoadmin:passwordone@csfle-
mongodb-<PETNAME>.mdbtraining.net:27017/
?replicaSet=rs0" --tls --tlsCAFile
/etc/pki/tls/certs/ca.cert
```

```
>
db.getSiblingDB("encryption").getCollect
ion("__keyVault").createIndex(
      {
          keyAltNames: 1
      },
      {
          unique: true,
          partialFilterExpression: {
               "keyAltNames": {
                    "$exists": true
      }}})
```

CSFLE Components - libmongocrypt

C library providing crypto and coordination with external components

- Performs encryption and decryption
- Caches Data Encryption Keys (DEKs)
- Caches Customer Master Keys (CMKs)
- Caches Collection information (JSONSchema)
- Orchestrates I/O between driver and Key Management Server (KMS)

libmongocrypt must be installed before CSFLE can be used

CSFLE Components - crypt_shared

Automatic Encryption Shared Library

Replacement for **mongocryptd** daemon

Dynamically loaded by MongoDB Driver

- Determines which fields to encrypt or decrypt
- Prevents client app from executing unsupported operations on encrypted fields

Does not perform any encryption or decryption itself.

CSFLE Components - mongocryptd

Daemon process for Automatic Encryption

Runs on the client, listens on port 27020

- Determines which fields to encrypt or decrypt
- Prevents client app from executing unsupported operations on encrypted fields
- Parses Encryption Schema and returns error if invalid

Only required for Automatic Encryption and if **crypt_shared** is not being used.

CSFLE Components - mongocryptd

mongocryptd does not perform any of the following:

- mongocryptd does not perform encryption or decryption itself
- mongocryptd does not access any encryption key material
- mongocryptd does not listen over the network

mongocryptd is available with MongoDB Enterprise or Atlas, any tier including free clusters. It is included in the MongoDB Enterprise Advanced server package. **mongocryptd** listens on port **27020** by default on localhost, it cannot listen on any other interface.

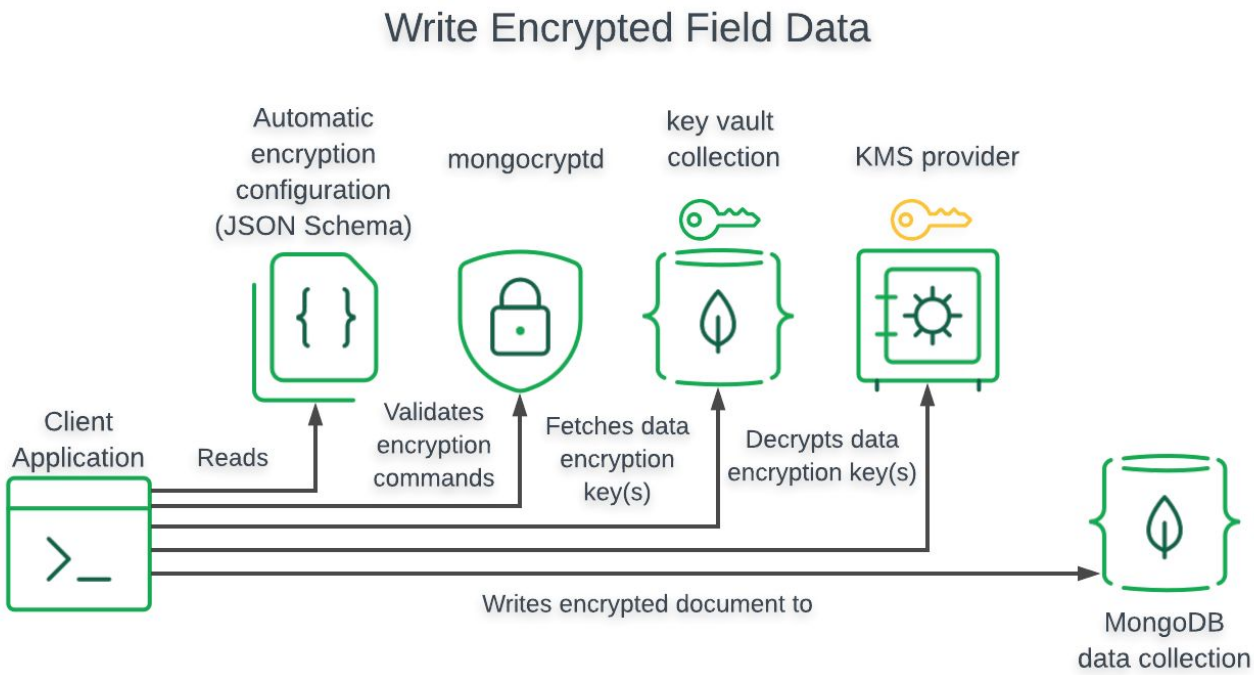
We recommend starting mongocryptd when the VM or container starts and disable the auto spawning options (this is discussed further in the workshop).

CSFLE Components - Driver

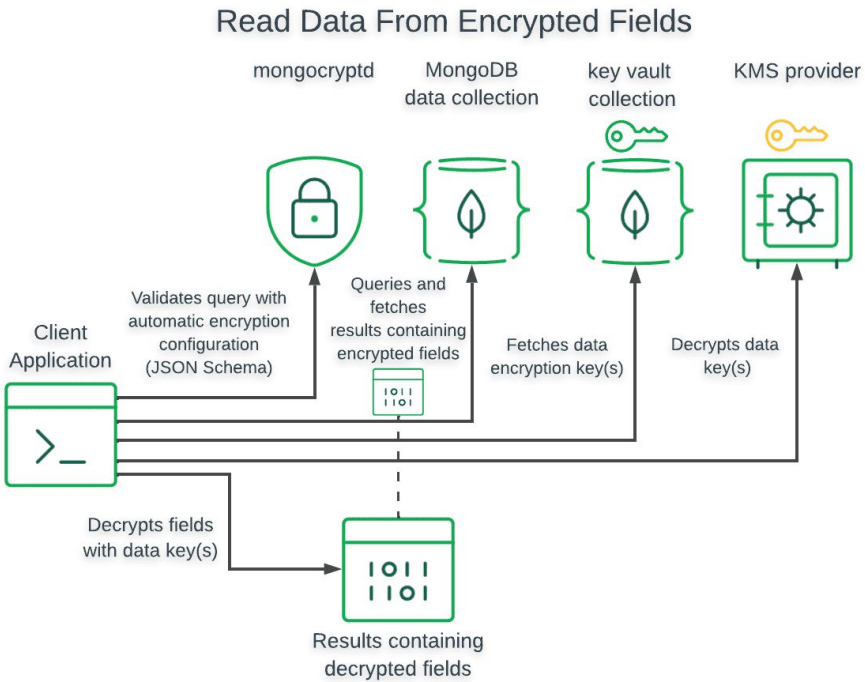
The **driver** is responsible for:

- performing all I/O needed at every stage:
  - running **listCollections** to return Encryption Schemas
  - communicating with **mongocryptd** or **crypt_shared**
  - fetching encrypted data keys (DEKs) from the Key Vault collection
  - sending encrypted DEKs to the KMS
  - receiving decrypted DEKs from the KMS.
- doing I/O asynchronously as needed.

CSFLE Details - Automatic Encryption



CSFLE Details - Automatic Decryption



Knowledge Check 1

#1. What is the Key Vault?

A	MongoDB Collection storing encrypted DEKs
B	External provider storing encryption keys
C	External provider storing CMKs
D	Decryption library running on port 27020
E	Don’t know

Answer in the next slide.

#1. What is the Key Vault?

A	MongoDB Collection storing encrypted DEKs
B	External provider storing encryption keys
C	External provider storing CMKs
D	Decryption library running on port 27020
E	Don’t know

#2. What does the Automatic Encryption Shared Library do?

A	Encrypts specific fields in a collection
B	Caches Data Encryption Keys
C	Identifies fields to automatically encrypt or decrypt
D	Decrypts commands sent to port 27020
E	Don’t know

Answer in the next slide.

#2. What does the Automatic Encryption Shared Library do?

A	Encrypts specific fields in a collection
B	Caches Data Encryption Keys
C	Identifies fields to automatically encrypt or decrypt
D	Decrypts commands sent to port 27020
E	Don't know

CSFLE Details

CSFLE Details - Encryption Algorithm

CSFLE uses encrypt-then-MAC Authenticated Encryption with Associated Data (AEAD) to allow integrity checking of encrypted and unencrypted portions of the payload. The encryption algorithm used is AES-256-CBC, , which is a symmetric encryption algorithm (magnitudes faster than asymmetric encryption), and the algorithm used for MAC generation is HMAC-SHA-512.

CSFLE provides two encryption types: deterministic or random. Deterministic encryption produces the same ciphertext for a given value every time it is encrypted whereas random encryption will produce different ciphertext every time a given value is encrypted. Fields that are encrypted deterministically can be queried because the ciphertext is the same every time the encryption engine is run, but fields encrypted with the random vector cannot because the ciphertext is different every time the encryption process is run.

CSFLE Details - Valid BSON Types

The following BSON types cannot be encrypted with deterministic encryption:

- Array
- Double and decimal128
- Object
- Bool
- javascriptWithScope (this is a deprecated type)

The following BSON types cannot be encrypted at all:

- Null
- Minkey
- Maxkey
- Undefined

CSFLE Details - Encryption Methods

There are two methods of encrypting and decrypting data using CSFLE with the driver API: manual (explicit) and/or automatic (implicit).

Manual encryption requires explicit commands to encrypt and decrypt data, and can be performed on almost any client that has access and credentials to the CMK. The DEK used for the encryption and decryption process is specified per field, per document (and therefore can be different for each field in any given document).

Manual (explicit) encryption does **not** require the **mongocryptd** or **crypt_shared** library. Automatic decryption is the default for all CSFLE clients and similarly does **not** require **monogocryptd** or **crypt_shared**. All encrypted payloads are returned to the driver with self-describing metadata about the DEK and encryption algorithm.

CSFLE Details - Automatic Encryption

Automatic encryption requires an Encryption Schema for all namespaces that use CSFLE. Each Encryption Schema details what fields to automatically encrypt, and for each field we specify the DEK UUID, the BSON type, and the encryption algorithm. The Encryption Schemas, key provider, and CMK are declared as an option with the MongoDB client.

Automatic encryption requires the **mongocryptd** or **crypt_shared** library to be installed, and the location can be specified as an additional option with the MongoDB client. The **mongocryptd** daemon can be spawned by the MongoDB client, or it can be started on boot. Starting it at boot is far less expensive than the driver starting **mongocryptd** or **crypt_shared**. If the mongocryptd is already running the MongoDB client may raise a warning, but will continue to operate correctly. The mongocryptd process MUST operate on the host with the MongoDB client.

CSFLE Details - Encryption Schema

The Encryption Schema is a JSON schema declared client side that is the same format as our JSON Schema Validator.

The Encryption Schema contains one object per namespace that uses CSFLE.

The Encryption Schema is used by **mongocryptd** or **crypt_shared** to mark fields that will be automatically encrypted with the associated DEK and algorithm; the mongocryptd does not perform any encryption, this is done by **libmongocrypt**.

The Encryption Schema, as a JSON Schema Validator, can also be declared server side as a backup and to prevent unencrypted data from being accidentally stored when either the client is configured incorrectly or not at all.

CSFLE Details - Encryption Schema

The client-side Encryption Schema cannot be used to perform other types of validation, which the server-side JSON Schema Validator can perform. If a client-side Encryption Schema exists the driver will not download the Encryption Schema from the server, unless the server has a Encryption Schema for a specific namespace the client does not have. If the client-side Encryption Schema is absent the driver downloads the server-side Encryption Schema.

The server-side checking using the JSON Schema Validator will only check if the highlighted fields in the schema are binary subtype 6. The server will not check the DEK or algorithm.

CSFLE Details - libmongocrypt

The MongoDB driver, via **libmongocrypt**, performs the encryption and decryption for both manual and automatic encryption and decryption. Installation or integration of **libmongocrypt** may be required for some drivers. Once again **libmongocrypt** MUST reside where the MongoDB client is running, we discourage running this on the MongoDB database server.

CSFLE Details - DEK generation

The DEKs can be created programmatically on-demand, which is useful where a new DEK is required for every new user or customer (such as to address GDPR). If your application does not require this functionality, the DEKs should be created outside your application and the application user's permissions should not have this privilege.

The DEKs are created by instantiating a **ClientEncryption** instance, which includes details of the key provider, the MongoDB Key Vault namespace, and the MongoClient object, and then referencing the CMK and key provider to use to wrap the DEK.

Key material can also be provided when creating a DEK, if this is not provided the **libmongocrypt** library will securely create this material.

DEKs can be destroyed using a **ClientEncryption** instance as well. Once again, if your application does not require this functionality this should not be allowed within the application or at the application user's permissions.

Key Management

Key Management

As highlighted, we store our encrypted DEKs within MongoDB database collection. This can be the same MongoDB as our encrypted data or this can be a separate MongoDB instance.

Remember we create a unique index on the collection for the **keyAltNames** field, which is the key alternative name (e.g. simplified name) field. We can search for DEKs by their alternative name, which makes things easier when we have hundreds or thousands of DEKs.

Why do you think we make this a unique index?

The database and collection names for the MongoDB Key Vault are only limited by the normal MongoDB restrictions. We recommend naming both something that is obvious and indicates they are for internal use.

ClientEncryption Class

All of our drivers have the **ClientEncryption** class, although this is not available directly in the Mongosh.

The **ClientEncryption** class has methods to:

- Create DEK
- Delete DEK
- Encrypt
- Decrypt
- Get DEK(s)
- Get DEK by alternative name
- Rewrap (encrypt) many DEKs
- Add DEK alternative name
- Remove DEK alternative name

As the **ClientEncryption** class does not have methods to manipulate collections, it cannot be used to set JSON Schema Validators or even create collections.

ClientEncryption Class

This class also has options to provide TLS settings for your chosen key provider, plus several other options.

In this course we will be setting the TLS options for our KMIP service, which will be language dependant.

Typed languages require setting the TLS options via method, and not just declaring the object.

Key Management - Key Provider

Each key provider has different attributes:

AWS - IAM User

```
"aws": {
   accessKeyId: string,
   secretAccessKey: string,
}
```

AWS - Assumed IAM Role

```
"aws": {
   accessKeyId: string,
   secretAccessKey: string,
   sessionToken: string // Required for temporary AWS credentials.
}
```

Or the following to take the credentials from the environment:

```
"aws': {}
```

Key Management - Key Provider

Each key provider has different attributes:

GCP - credentials

```
"gcp": {
   email: string,
   privateKey: byte[] | string, // bytes or base64 encoded string.
   endpoint: string, // Default: oauth2.googleapis.com
}
```

GCP - access token

```
"gcp": {
   accessToken: string
}
```

Key Management - Key Provider

Each key provider has different attributes:

Azure

```
"azure": {
   tenantId: string,
   clientId: string,
   clientSecret: string,
   identityPlatformEndpoint: string // Default: login.microsoftonline.com
}
```

Key Management - Key Provider

Each key provider has different attributes:

KMIP

```
"kmip": {
   endpoint: string,
}
```

The KMIP provider will also require some TLS options

Local

```
"local" {
   key: byte[96] | string // The master key used to encrypt/decrypt DEKs. May be passed
as a base64 encoded string.
}
```

Key Management - DEK creation

Mongosh does not use the ClientEncryption class directly unlike the driver APIs. With mongosh you create an encrypted client and then use the **keyvault** method.

An encrypted MongoClient can be created by providing an **AutoEncryptionOpts** options object when instantiated the MongoClient. Some drivers will have a different name for the **AutoEncryptionOpts** object, such as **AutoEncryptionSettings** in Java. The AutoEncryptionsOpts contains information about the key provider, Encryption Schema, and MongoDB Key Vault (plus a few other options we do not require as yet).

When we create the encrypted MongoClient in the Mongosh we will not provide any options for autoencryption or the mongocryptd. We basically use the encrypted MongoClient in Mongosh to get the ClientEncryption object.

Key Management - DEK creation

The **ClientEncryption** object has a method to create DEKs called **createKey**. This requires the name of the key provider, the id or name of the CMK, and the alternative name of the DEK.

Each key provider has different attributes for the CMK:

```
AWS
{
 "key": "<ID OF CMK>",
 "region": "<AWS REGION FOR KMS>"
}

Azure
{
 "keyName": "<Name of the CMK>",
 "keyVaultEndpoint": "<URL of the azure key vault>"
}
```

Key Management - DEK creation

Each key provider has different attributes for the CMK:

```
GCP
{
 "projectId": "<ID of the project>",
 "location": "<Region of the CMK>",
 "keyRing": "<ID for the group of keys the CMK belongs to>",
 "keyName": "<ID of the CMK>",
}

KMIP
{
 "keyId": "<keyId field of the 96 byte secret data object>",
 "endpoint": "<URI of the KMIP provider>"
}

Local
{}
```

ClientEncrypt - Mongosh

```
// start mongosh with `mongosh --nodb`
const provider = {
 "kmip": { // <-- KMS provider name
    "endpoint": "csfle-kmip-<PETNAME>.mdbtraining.net"
 }
}

const tlsOptions = {
  kmip: {
    tlsCAFile: "/etc/pki/tls/certs/ca.cert",
    tlsCertificateKeyFile: "/home/ec2-user/server.pem"
  }
}

const autoEncryptionOpts = {
 kmsProviders : provider,
 schemaMap: {}, //no Encryption Schema
 keyVaultNamespace: "__encryption.__keyVault",
 tlsOptions: tlsOptions
}
```

Create DEK - Mongosh

```
encryptedClient =
Mongo("mongodb://mongoadmin:passwordone@csfle-mongodb-<PETNAME>.mdbtraining.net:27017/?replicaSe
t=rs0&tls=true&tlsCAFile=%2Fetc%2Fpki%2Ftls%2Fcerts%2Fca.cert", autoEncryptionOpts)

keyVault = encryptedClient.getKeyVault()

keyVault.createKey(
 "kmip", // <-- KMS provider name
 {
   "keyId": "1"
 }, // <-- CMK info (specific to KMIP in this case)
 ["dataKey1"] // <-- Key alternative name
)

// Retrieve all the keys
keyVault.getKeys()
```

Create DEK - Mongosh Exercise

We have a KMIP device in our environments on the second instance (csfle-kmip-**PETNAME**.mdbtraining.net).

We already have a CMK created in the device and the `keyId` is 1.

Modify the code in the previous slides to use the KMIP and create a single DEK.

This should be performed on your client (csfle-3 instance) via the mongosh.

(the commands can be found in the commands.txt file within the csfle_workshop_skeleton directory)

Key Management

We will use an user with elevated privileges to create our DEK because were initially have a simple use case.

The best practise is to not give user privileges to the key vault unless they need to perform specific tasks. In this case our application user only needs to perform **find** on the key vault to find and retrieve the encrypted DEKs.

Create Role - Mongosh

Because we love security in MongoDB we are going to create a custom role that will allow the application user to only perform find operations on our MongoDB Key Vault.

We need to disconnect (or start a new session) our encrypted mongosh session:

1. Disconnect from the monosh via **exit**// use a normal (unencrypted) mongosh
2. Connect to MongoDB via an unencrypted mongosh:

```
mongosh
"mongodb://mongoadmin:passwordone@csfle-mongodb-<PETNAME>.mdbtraining.net:27017/?replicaSet=rs0&tls=true&tlsCAFile=%2Fetc%2Fpki%2Ftls%2Fcerts%2Fca.cert"
```

Create Role - Mongosh

Because we love security in MongoDB we are going to create a custom role that will allow the application user to only perform find operations on our MongoDB Key Vault.

```
// use a normal (unencrypted) mongosh
use admin;
db.createRole({
 "role": "cryptoClient",
 "privileges": [
   {
      resource: {
        db: "__encryption",
        collection: "__keyVault"
      },
      actions: [ "find" ]
    }
  ],
  "roles": [ ]
})
```

Create User - Mongosh

Because we love security in MongoDB we are going to create a custom role that will allow the application user to only perform find operations on our MongoDB Key Vault.

```
// use a non-encrypted mongosh
use admin;
db.createUser({
 "user": "app_user",
 "pwd": "password123", // this is bad practice!
 "roles": ["cryptoClient", {'role': "readWrite", 'db': 'companyData'} ]
})
```

We are going to use the companyData database for all our documents

Create Role & User - Mongosh Exercise

Using the mongosh, create the role and user as described in the previous slides.

Our role only needs find on the __encryption.__keyvault namespace, plus the readWrite role on the companyData database

Remember to use the csfle-3 VM and that the MongoDB replica set has authentication and TLS enabled.

The password is password123 if you use the illustrated command.

(the commands can be found in the commands.txt file within the csfle_workshop_skeleton directory)

Manual Encryption and Decryption

Drivers

We are now moving away from the mongosh for most of this session and will be coding in the programming language of your choice utilising the MongoDB drivers to perform the MongoDB tasks.

Manual Encryption - Steps

1. Create a DEK or DEKs (we have done this already)
2. Create a **ClientEncryption** with the MongoDB client, key provider details, key vault namespace, and other options as required
3. Retrieve the **UUIDs** of the required DEKs using the **ClientEncryption** via the DEK alternative name method (depending on language)
4. Explicitly call ClientEncryption.encrypt method with the value to encrypt, the algorithm, and DEK UUID
5. Use the encrypted values in your write operation via a normal MongoDB client

Manual Decryption - Steps

1. Perform steps 1 to 3 from the encryption steps
2. Explicitly call ClientEncryption.encrypt method for each query value with the value to encrypt, the algorithm, and DEK UUID
3. Use the encrypted values in your query operation via a normal MongoDB client
4. Retrieve the document and call the ClientEncryption.decrypt method with the encrypted value for each encrypted value to be decrypted.
5. Substitute the encrypted values for the decrypted values

Libraries

```
# Python

from pymongo.errors import EncryptionError
from bson.codec_options import CodecOptions
from pymongo.encryption import Algorithm
from bson.binary import STANDARD
from pymongo.encryption import ClientEncryption
From pymongo import MongoClient

// Go

"go.mongodb.org/mongo-driver/bson"
"go.mongodb.org/mongo-driver/bson/primitive"
"go.mongodb.org/mongo-driver/mongo"
"go.mongodb.org/mongo-driver/mongo/options"

Go must be run as follows to ensure CSFLE is enabled:

go build -tags cse [project_name]
go run -tags cse [project_name]
```

Libraries

```
# Java

import com.mongodb.ClientEncryptionSettings;
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.model.vault.DataKeyOptions;
import com.mongodb.client.vault.ClientEncryption;
import com.mongodb.client.vault.ClientEncryptions;
import org.bson.BsonBinary;
import org.bson.BsonDocument;
import org.bson.Document;
```

TLS & Java

```
# Because Java has a "unique" way to handle TLS certificates and key, you will need
the following if using Java (from command line; use “mongodb” as the password when
prompted):

openssl pkcs12 -export -in server.pem -out keystore.p12

# Convert PKCS12 to JKS keystore
keytool -importkeystore -destkeystore keystore.jks -srckeystore keystore.p12
-srcstoretype pkcs12

# Import the CA certificate into the truststore
keytool -importcert -trustcacerts -file /etc/pki/tls/certs/ca.cert -keystore
truststore.jks -storepass mongodb

# Then from ~/ with properties baked in you can compile and run:
# mvn -f pom_<dir_name>.xml compile
# java -cp "./target/classes:/home/ec2-user/lib/*" App
```

ClientEncryption Class - Python

```
# AWS user

provider = "aws"
kms_provider = {
  provider: {
    "accessKeyId": "<IAM User Access Key ID>",
    "secretAccessKey": "<IAM User Secret Access Key>",
  }
}
keyvault_namespace = f"__encryption.__keyVault" # can be any valid namespace

# instantiate the object
client_encryption = ClientEncryption(
   kms_provider,
   keyvault_namespace,
   Client, # MongoClient previously instantiated
   CodecOptions(uuid_representation=STANDARD)
)
```

ClientEncryption Class - Go

```
# AWS IAM Role

kp = map[string]map[string]interface{}{
     "aws": {
          "accessKeyId":     accessKeyId,
          "secretAccessKey": secretAccessKey,
          "sessionToken":    sessionToken,
     },
}

// Keyvault namespace
kns := "__encryption.__keyVault"

// ClientEncryption options
o := options.ClientEncryption().SetKeyVaultNamespace(kns).SetKmsProviders(kp)

// instantiate the ClientEncryption object
clientEncryption, _ := mongo.NewClientEncryption(c, o)
```

ClientEncryption Class - Java

```
String provider = "aws";

Map<String, Map<String, Object>> kmsProvider = new HashMap<String, Map<String,
Object>>();
Map<String, Object> awsProviderInstance = new HashMap<String, Object>();
        awsProviderInstance.put("accessKeyId", credentials.accessKeyId());
        awsProviderInstance.put("secretAccessKey", credentials.secretAccessKey());
        awsProviderInstance.put("sessionToken", credentials.sessionToken());
        kmsProvider.put(provider, awsProviderInstance);

MongoNamespace keyvaultNamespace = new MongoNamespace("__encryption.__keyVault");

ClientEncryptionSettings encryptionSettings = ClientEncryptionSettings.builder()
        .keyVaultMongoClientSettings(MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString(connectionString))
            .uuidRepresentation(UuidRepresentation.STANDARD)
            .build())
        .keyVaultNamespace(keyvaultNamespace)
        .kmsProviders(kmsProviders)
        .build();
ClientEncryption clientEncryption = ClientEncryptions.create(encryptionSettings);
```

Manual Encryption - Exercise

Using your language of choice, review the code that instantiates the ClientEncryption instance. Because we are using a KMIP device we have more options to provide, which has already been done.

(Skeleton code is in the manual_encryption directory for this exercise on the client VM, which is under /home/ec2-user/csfle_workshop_skeleton)

Attributes you might need:

```
CA certificate location: /etc/pki/tls/certs/ca.cert

PEM file location: /home/ec2-user/server.pem

Username: app_user

Password: <YOUR_SELECTED_PASSWORD>
```

You only need to update the PETNAME and the MDB_PASSWORD variables for this step, but review the code to see what we have provided.

Manual Encryption - Exercise

Because we are using a KMIP device, we need to provide a CA certificate and the certificate key file to use for authentication and TLS with the KMIP. This will be part of the ClientEncryption objects, as such:

```
client_encryption = ClientEncryption(
  kms_provider, // variable of the key provider
  keyvault_namespace, // variable of the keyvault namespace string
  client, // MongoDB client object
  CodecOptions(uuid_representation=STANDARD),
  kms_tls_options = {
   "kmip": {
     "tlsCAFile": "/etc/pki/tls/certs/ca.cert",
     "tlsCertificateKeyFile": "/home/ec2-user/server.pem"
   }
  }
)
```

ClientEncryption Class - Encrypt

Once the instant is instantiated, values can be encrypted (or decrypted) by calling the encrypt (or decrypt) method with the DEK and algorithm.

We can retrieve our DEK UUID via the ClientEncryption instance or a normal MongoDB client. If using the ClientEncryption instance, we can call the **get_key_by_alt_name** method and select just the _id, which is the DEK's UUID. We then use this when we encrypt our data.

The method to manually encrypt a value is:

```
<ClientEncrytion_Instance>.encrypt(Value_to_encrypt, Algorithm, DEK UUID)
```

Each language will have a constant for the algorithm.

ClientEncryption Class - Python

```
encrypted_name = client_encryption.encrypt(
   "Ivan",
   Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
   key_id = data_key_id_1, # DEK UUID
)
encrypted_foods = client_encryption.encrypt(
   ["Pizza", "Beer", "Salad"],
   Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Random,
   key_id = data_key_id_2, # DEK UUID
)
coll.insert_one({"name": encrypted_name, "foods": encrypted_foods})
```

ClientEncryption Class - Go

```
encryptionOpts := options.Encrypt().
            SetAlgorithm("AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic").
            SetKeyID(<DEK_UUID>)
rawValueType, rawValueData, _ := bson.MarshalValue("Ivan")
rawValue := bson.RawValue{Type: rawValueType, Value: rawValueData}
encryptionOpts := options.Encrypt().
            SetAlgorithm("AEAD_AES_256_CBC_HMAC_SHA_512-Random").
            SetKeyID(<DEK_UUID>)
rawValueType, rawValueData, _ := bson.MarshalValue([]string{"Pizza", "Beer", "Salad"})
rawValue := bson.RawValue{Type: rawValueType, Value: rawValueData}
encrypted_food, _ := clientEncryption.Encrypt(context.TODO(), rawValue, encryptionOpts)
result, _ := coll.InsertOne(context.TODO(), bson.M{"name": encrypted_name"foods":
encrypted_foods})
```

ClientEncryption Class - Java

```
encryptionOpts = EncryptOptions("AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic")
           .keyId(new BsonBinary(UUID.fromString(<DEK_UUID>)));

BsonBinary encryptedName = clientEncryption.Encrypt(new BsonString("Ivan"),
encryptionOpts)

encryptionOpts = EncryptOptions("AEAD_AES_256_CBC_HMAC_SHA_512-Random")
           .keyId(new BsonBinary(UUID.fromString(<DEK_UUID>)));

BsonBinary encryptedFoods = clientEncryption.Encrypt(new
BsonArray().parse("[\"Pizza\", \"Beer\", \"Salad\"]"), encryptionOpts)

collection.insertOne(new Document("name", encryptedName).append("foods",
encryptedFoods));
```

Manual Encryption Exercise

Now enhance your code to insert the document that is hardcoded in your code. We have only one DEK, which is easy for us, but we will use different modes of encryption depending on the field. We will insert the document into the `companyData.employee` namespace.

The following fields will use deterministic encryption:

- name.firstName
- name.lastName

The following fields will use random encryption:

- name.otherNames
- address (encryption the whole object)
- dob
- phoneNumber
- salary
- taxIdentifier

Manual Encryption Exercise

Now use the mongosh without an encrypted client to look at the data you have inserted into your `companyData.employee` collection.

You should have confidence the data is encrypted.

The issue here is that we have nothing preventing data going into this collection in unencrypted format, but we will fix that soon.

Encrypted Data

The encrypted data is a BSON Binary subtype 6, which will look similar to:

Python:

```
BinData(6,"AgWdqufm/0qQn4CtIBH3L5ICaEYhCpO+fkB4IPg1kAWgOfvuSjWgqOPRc2EVCCyhzgrmyv0hPCnaAd
Dj7XC2MnzvsoszeQuXlktYDIgTVoRXXQ==")
```

Mongosh:

```
Binary(Buffer.from("014fdb8e3dcc23422dafbd9537400dda1a028f4b50102e34d1c1f113715859c7db9ea
2a29e0ad5ed5ed793ebe7815060be12c0ebd1e41e926694425a58f6face698d9a68f2a9b9e8b8f5ffb31bebdf
dd7d3f", "hex"), 6)
```

The data contains information about the DEK and algorithm (1 is deterministic and 2 is random) used to encrypt the data, plus the ciphertext:

```
// Structure of an encrypted field
struct {
   uint8 subtype = (1 or 2); <-- Algorithm used
   uint8 key_uuid[16]; <-- DEK UUID
   uint8 original_bson_type;
   uint8 ciphertext[ciphertext_length];
}
```

ClientEncryption Class - Decrypt

To manually query an encrypted field, the value for the field we are going to query would need to be manually encrypted.

For each encrypted field value in the returned document the **decrypt** method would need to be called on our ClientEncryption instance and as the DEK UUID and algorithm are included in the encrypted data we do not need to provide the DEK or algorithm in the **decrypt** method call.

Note that we **could** do a range or substring search with manual encryption, but this makes no sense because the data is binary, and queries would fail to return meaningful results. Only equality queries makes sense with CSFLE.

The method to manually decrypt a value is:

```
<ClientEncrytion_Instance>.decrypt(Value_to_decrypt)
```

ClientEncryption Class - Python

```
encrypted_name = client_encryption.encrypt(
   "Ivan",
   Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
   key_id = data_key_id_1, # DEK UUID
)
encrypted_doc = coll.find_one({"name": encrypted_name})

decrypted_doc = {}
decrypted_doc["name"] = client_encryption.decrypt(encrypted_doc["name"])
decrypted_doc["foods"] = client_encryption.decrypt(encrypted_doc["foods"])

print(f"Decrypted document: {decrypted_doc}")
```

ClientEncryption Class - Go

```
encryptionOpts := options.Encrypt().
            SetAlgorithm("AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic").
            SetKeyID(<DEK_UUID>)

rawValueType, rawValueData, _ := bson.MarshalValue("Ivan")
rawValue := bson.RawValue{Type: rawValueType, Value: rawValueData}
encryptedName, _ := clientEncryption.Encrypt(context.TODO(), rawValue, encryptionOpts)

_ := collection.findOne(bson.M{"name": encryptedName}).Decode(&result)

result["name"], _ = clientEncryption.Decrypt(context.TODO(),
result["name"].(primitive.Binary))
result["foods"], _ = clientEncryption.Decrypt(context.TODO(),
result["foods"].(primitive.Binary))

fmt.Printf("Decrypted document: %s\n", result)
```

ClientEncryption Class - Java

```
encryptionOpts = EncryptOptions("AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic")
            .keyId(new BsonBinary(UUID.fromString(<DEK_UUID>)));

BsonBinary encryptedName = clientEncryption.encrypt(new BsonString("Ivan"),
encryptionOpts);
Document result = collection.find(eq("name": encryptedName)).first();

result.replace("name", clientEncryption.decrypt(new
BsonBinary(BsonBinarySubType.ENCRYPTED, result.get("name", Binary.class).getData())));
result.replace("foods", clientEncryption.decrypt(new
BsonBinary(BsonBinarySubType.ENCRYPTED, result.get("foods", Binary.class).getData())));

System.out.println("Decrypted document: " + result.toJson());
```

Manual Decryption Exercise

Enhance your code once again to query the `companyData.employee` namespace for the field `name.firstName` with the value "`Kuber`". You need to encrypt the field for the query. We have included code to manually decrypt all the encrypted fields in the returned document. The skeleton code for this is in the `manual_decryption` directory

Remember you must manually encrypt the query filter value.

Manual Encryption and Decryption

As can be seen, explicitly encrypting and decrypting data does involve encrypting and decrypting each field, including any field that we query that is encrypted in the database.

Because the DEK UUID and encryption algorithm is included in the encrypted data the automatic decryption function can be used instead of manual decryption: using this is all dependent on the use case. Using automatic decryption will be explained in the next section.

So why would we use manual encryption and decryption?

Automatic Encryption and Decryption

Automatic Encryption and Decryption

To perform automatic encryption **mongocryptd** or **crypt_shared** are required. The MongoClient can spawn these processes. If using mongocryptd it is better to have it start on boot if encryption is a regular task.

Mongocryptd listens on port 27020 by default on localhost: the port number can be modified when creating the encrypted MongoClient, but not the interface (e.g. localhost)

Remember that mongocryptd and crypt_shared marks fields to be encrypted from the specifications in the Encryption Schema, they do not perform the actual encryption.

mongocryptd is part of the enterprise server package and can be installed manually from this package if desired.

crypt_shared is a separate dynamic library that can be installed manually as well.

Encrypted Client

As highlighted, the MongoClient will spawn the mongocryptd process if required, this can be disabled.

The MongoClient can be provided with an Auto Encryption Options object to enable encryption. This requires the following:

- Key providers (aws, gcp, azure, kmip, and/or local)
- Keyvault namespace (as a string)
- Encryption Schema supplied as an object to the schema_map variable (example to follow)

There are several other options as well:

- Disable auto encryption (auto decryption still remains enabled)
- Keyvault MongoClient if the keyvault is on a different MongoDB instance to the encrypted client
- Bypass spawning mongocryptd
- URI of mongocryptd if not using the default port (27020)
- Key provider TLS options
- Force use of crypt_shared
- Crypt_shared path
- Mongocryptd path

Encrypted Client

Before we can automatically encrypt anything we need either the **mongocryptd** package installed or **crypt_shared** package on our client.

We will use the **crypt_shared** package. We have installed this for you in **/lib/mongo_crypt_v1.so**

We declare the absolute path for **crypt_shared** in our **automation_opts** object via **crypt_shared_lib_path** (specific to language).

If you forget to install either of these you will get an exception (this is the missing mongocryptd error):

```
pymongo.errors.ServerSelectionTimeoutError: localhost:27020: [Errno 111]
Connection refused, Timeout: 10.0s, Topology Description: <TopologyDescription
id: 63991d9b816bf469c6b7a58e, topology_type: Unknown, servers:
[<ServerDescription ('localhost', 27020) server_type: Unknown, rtt: None,
error=AutoReconnect('localhost:27020: [Errno 111] Connection refused')>]>
```

Encrypted Client - Libraries

```
# Python

from pymongo.errors import EncryptionError
from bson.codec_options import CodecOptions
from bson.binary import STANDARD
from pymongo.encryption_options import AutoEncryptionOpts
from pymongo import MongoClient

// Go

"go.mongodb.org/mongo-driver/bson"
"go.mongodb.org/mongo-driver/bson/primitive"
"go.mongodb.org/mongo-driver/mongo"
"go.mongodb.org/mongo-driver/mongo/options"
```

Libraries

```
# Java

import com.mongodb.ClientEncryptionSettings;
import com.mongodb.AutoEncryptionSettings;

import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.model.vault.DataKeyOptions;
import com.mongodb.client.vault.ClientEncryption;
import com.mongodb.client.vault.ClientEncryptions;
import org.bson.BsonBinary;
import org.bson.BsonDocument;
import org.bson.Document;
```

Encrypted Client - Python

```
# AWS IAM Assumed Role
provider = "aws"
kms_provider = {
  provider: {
    "accessKeyId": "<Assumed IAM Role Access Key ID>",
    "secretAccessKey": "<Assumed IAM Role Secret Key>"
  }
}

keyvault_namespace = f"__encryption.__keyVault"

auto_encryption = AutoEncryptionOpts(
  kms_provider,
  keyvault_namespace,
  schema_map = schema_map,
  key_vault_client = <MONGOCLIENT_TO_SEPARATE_KEYVAULT>, # optional
  mongocryptd_spawn_path="<PATH_TO_MONGOCRYPTD>" # optional
)
```

Encrypted Client - Python

```
client = pymongo.MongoClient(
  <CONNECTION_STRING>,
  serverSelectionTimeoutMS=<TIMEOUT_VALUE>,
  tls=True,
  tlsCAFIle="<PATH_TO_CA_PEM>",
  auto_encryption_opts=auto_encryption_opts
)
```

Encrypted Client - Go

```
// IAM Role
kp = map[string]map[string]interface{}{
     "aws": {
          "accessKeyId":     accessKeyId,
          "secretAccessKey": secretAccessKey,
          "sessionToken":    sessionToken,
     },
}

// Keyvault namespace
kns := "__encryption.__keyVault"

autoEncryptionOpts := options.AutoEncryption().
          SetKeyVaultNamespace(kns).
          SetKmsProviders(kp).
          SetSchemaMap(schemaMap)
```

Encrypted Client - Go

```
c := "<CONNECTION_STRING>"

client, err := mongo.Connect(
  context.TODO(),
  options.Client().ApplyURI(c).SetAutoEncryptionOptions(autoEncryptionOpts)
)
```

Encrypted Client - Java

```
// AWS User
String provider = "aws";
Map<String, Map<String, Object>> kmsProvider = new HashMap<String, Map<String,
Object>>();
Map<String, Object> awsProviderInstance = new HashMap<String, Object>();
awsProviderInstance.put("accessKeyId", credentials.accessKeyId());
awsProviderInstance.put("secretAccessKey", credentials.secretAccessKey());
kmsProvider.put(provider, awsProviderInstance);


// Keyvault namespace
MongoNamespace keyvaultNamespace = new MongoNamespace("__encryption.__keyVault");
```

Encrypted Client - Java

```
AutoEncryptionSettings autoEncryptionSettings = AutoEncryptionSettings.builder()
            .keyVaultNamespace(keyvaultNamespace.getFullName())
            .kmsProviders(kmsProvider)
            .schemaMap(schemaMap)
            .build();

ConnectionString mdbConnectionString = new ConnectionString(connectionString);

MongoClientSettings.Builder settingsBuilder = MongoClientSettings.builder()
    .applyConnectionString(mdbConnectionString)
    .uuidRepresentation(UuidRepresentation.STANDARD)
    .autoEncryptionSettings(autoEncryptionSettings)
    .applyToSslSettings(builder -> {
        builder.enabled(true);
    })
    .build();
```

Encrypted Client - Java

```
MongoClientSettings settings = settingsBuilder.build();
MongoClient mongoClient = MongoClients.create(settings);
```

Automatic decryption of data

Because the encrypted fields contain the DEK UUID and algorithm information use the encrypt the field, the client can decrypt the fields automatically if using an encrypted MongoClient because all the information the client needs to decrypt is in the field itself (as long as the client has access to the CMK and Key Provider). This means that even when manually encrypting fields, you can automatically decrypt the fields.

Manual Encryption and Automatic Decryption Exercise

In the man_encryption_auto_decryption directory, look at the code.
Enhance the code to perform a findOne/find_one to find the user "Poorna".

Attempt to query the salary field and see what happens.

Do you think the aggregation framework works with encrypted data? If so, what limitations?

A Encryption Schema (which we will discussed shortly) is not required for automatic decryption, therefore you declare the schemaMap option for your automatic encryption options object as an empty dictionary/object/hash and set the bypassAutoEncryption option to TRUE

Encryption Schema

The JSON Encryption Schema is a strict subset of the JSON Schema Draft 4 standard syntax with two additional keywords specific for MongoDB : **encrypt** and **encryptMetadata**. The JSON Encryption Schema is constructed exactly like a Encryption Schema that would be used by the JSON Schema Validator for a collection, but with an key for each namespace that will have encrypted fields.

The **encrypt** object is used to describe the algorithm (**algorithm**), BSON type (**bsonType**), and DEK ID (**keyId**) for a specific field. Remember the limitations of which BSON types cannot be encrypted with deterministic encryption and which BSON types cannot be encrypted at all. Use the **bsonType** attribute to describe what is the BSON type of field and the **algorithm** attribute to select either deterministic or random encryption. The **keyId** is the UUID of the DEK or a JSON pointer to a field within the document that contains the DEK UUID.

If the **encryptMetadata** object is declared in the parent object and all child objects under that parent will have the specified attributes used to encrypt the fields, this could be just the DEK ID or the algorithm, or both: each field that requires encryption must still have the **encrypt** keyword with the **bsonType** attribute for the field.

Encryption Schema

You also **cannot** put check constraints on encrypted fields, such as checking if values are greater than a certain value.

The Encryption Schema can contain one or more keys that are database & collection namespaces.

The Encryption Schema is only required for encryption, not for decryption.

At this point, think about what the Encryption Schema implies, both for client and server. We need extra planning up front when using CSFLE because it is more challenging to change the schema once we have one in place because we have applied the encryption already and changing DEKs will prevent us from querying values that already exist (unless we perform manual encryption with the old DEK or re-encrypt current documents).

The schemas are language-native structures or string literals that is parsed as JSON or unmarshal in JSON. Our slides are in language-native structures for typed languages, but the exercises use string literals for ease and speed.

Encryption Schema - Python

```
schema_map = {
  "<DATABASE>.<COLLECTION>": {
   "bsonType": "object",
   "properties": {
     "<FIELD NAME>": {
       "encrypt": {
         "keyId": [UUID(<UUID_OF_KEY>)],
         "bsonType": "int",
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic",
       }
     },
   },
...
}
```

Encryption Schema - Python

```
schema_map = {
  "<DATABASE>.<COLLECTION>": {
   "bsonType": "object",
   "encryptMetadata": {
     "keyId": [UUID(<UUID_OF_KEY>)],
     "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic",
   },
   "properties": {
     "<FIELD NAME>": {
       "encrypt": {
         "bsonType": "array",
       }
     },
   },
...
}
```

Encryption Schema - Python

```
schema_map = {
  "<DATABASE>.<COLLECTION>": {
   "bsonType": "object",
   "encryptMetadata": {
     "keyId": [UUID(<UUID_OF_KEY>)],
     "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic",
   },
   "properties": {
     "<FIELD NAME>": {
       "encrypt": {
         "bsonType": "array",
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random", //overriding the algorithm
       }
     },
   },
...
}
```

Encryption Schema - Go

```
schemaMap = bson.M{
 "<DATABASE>.<COLLECTION>": bson.M{
   "bsonType": "object",
   "properties": bson.M{
     "<FIELD NAME>": bson.M{
       "encrypt": bson.M{
         "keyId": bson.A{
           <DEK_UUID>,
         },
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
         "bsonType": "string",
       },
     },
...
}
```

Encryption Schema - Go

```
schemaMap = bson.M{
 "<DATABASE>.<COLLECTION>": bson.M{
   "bsonType": "object",
   "encryptMetadata": bson.M{
     "keyId": bson.A{
       <DEK_UUID>,
     },
     "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
   },
   "properties": bson.M{
     "<FIELD NAME>": bson.M{
       "encrypt": bson.M{
         "bsonType": "string",
       },
     },
...
}
```

Encryption Schema - Go

```
schemaMap = bson.M{
 "<DATABASE>.<COLLECTION>": bson.M{
   "bsonType": "object",
   "encryptMetadata": bson.M{
     "keyId": bson.A{
       <DEK_UUID>,
     },
     "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
   },
   "properties": bson.M{
     "<FIELD NAME>": bson.M{
       "encrypt": bson.M{
         "bsonType": "string",
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic",
       },
     },
...
```

Encryption Schema - Java

```
String dekId = "<DEK_UUID>";
Document jsonSchema = new Document().append("bsonType", "object")
    .append("properties", new Document()
        .append("<FIELD NAME>", new Document().append("encrypt",
            new Document().append("keyId",
                 new ArrayList<>((Arrays.asList(new Document().append("$binary",
                      new Document()
                           .append("base64", dekId)
                           .append("subType", "04")))))))
            .append("bsonType", "string")
            .append("algorithm", "AEAD_AES_256_CBC_HMAC_SHA_512-Random")))
        ....);
HashMap<String, BsonDocument> schemaMap = new HashMap<String, BsonDocument>();
schemaMap.put("<DATABASE>.<COLLECTION>", BsonDocument.parse(jsonSchema.toJson()));
```

Encryption Schema - Java

```
String dekId = "<DEK_UUID>";
Document jsonSchema = new Document().append("bsonType", "object").append("encryptMetadata",
       new Document().append("keyId", new ArrayList<>((Arrays.asList(
           new Document().append("$binary", new Document()
                .append("base64", dekId)
                .append("subType", "04")))))))
            .append("algorithm", "AEAD_AES_256_CBC_HMAC_SHA_512-Random")))
    .append("properties", new Document()
        .append("<FIELD NAME>", new Document().append("encrypt", new Document()
            .append("bsonType", "string")
        ....);
HashMap<String, BsonDocument> schemaMap = new HashMap<String, BsonDocument>();
schemaMap.put("<DATABASE>.<COLLECTION>", BsonDocument.parse(jsonSchema.toJson()));
```

Encryption Schema - Java

```
String dekId = "<DEK_UUID>";
Document jsonSchema = new Document().append("bsonType", "object").append("encryptMetadata",
       new Document().append("keyId", new ArrayList<>((Arrays.asList(
           new Document().append("$binary", new Document()
                .append("base64", dekId)
                .append("subType", "04")))))))
            .append("algorithm", "AEAD_AES_256_CBC_HMAC_SHA_512-Random")))
   .append("properties", new Document()
       .append("<FIELD NAME>", new Document().append("encrypt", new Document()
           .append("bsonType", "string")
           .append("algorithm", "AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic"))
       ....);
HashMap<String, BsonDocument> schemaMap = new HashMap<String, BsonDocument>();
schemaMap.put("<DATABASE>.<COLLECTION>", BsonDocument.parse(jsonSchema.toJson()));
```

Encryption Schema - Client and Server

If the Encryption Schema is not declared with the MongoClient options, the MongoClient will attempt to download a Encryption Schema from the server.

If the client has a Encryption Schema that is valid but is missing the key for a specific namespace and the server has a schema for that specific namespace, the client will download the missing schema for that specific namespace from the server.

Therefore, if the server has schemas (JSON Schema Validators) for namespaces **database0.collection0**, **database0.collection1**, and **database1,collection5**, but the client only has keys within the Encryption Schema for namespaces **database0.collection0** and **database0.collection1**, the client will download the schema for **database1.collection5** from the server, but not the other two namespaces.

The server will still check the payload if there is a server-side schema on collections, but only checks the highlighted fields are binary subtype 6 BSON type, the server does not check which DEK or algorithm is used.

Encryption Schema - Client and Server

Having a server-side schema on the collections means that the client-side Encryption Schema needs to be updated to include the keys for the missing namespaces to ensure that the client can properly decrypt the encrypted fields in those namespaces, and so the server-side schema acts as an up-to-date safety net. It is important to ensure that the Encryption Schema is consistent between the client and server to avoid errors (incorrect DEKs or algorithms) and to ensure that the encrypted fields are encrypted with the correct DEKs and algorithms, and can be queried.

Having a server-side schema validator acts as a safety net so unencrypted data is not accepted by the server. The server-side schema validator can be used with **automatic** or **manual** encryption methods, therefore the schema can assist even when manually encrypting.

MongoDB recommends that you always use a server-side schema validator.

Encryption Schema - Set for server - Mongosh

Use the Encryption Schema as a JSON Schema Validator when creating the collection:

```
db.getSiblingDB("companyData").createCollection(
  "employee",
  {
    "validator": {
      "$jsonSchema": { // Note the namespace is not here, unlike the Encryption Schema
        "bsonType": "object",
        "encryptMetadata": {
          "keyId": [UUID("<UUID_OF_KEY>")],
          "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
        },
        "properties": {
          "<FIELD NAME>": {
....
  }
)
```

Encryption Schema Exercise - Background

You are going to use automatic encryption for your documents in your **employee** collection inside your **companyData** database.

You want to use deterministic encryption for two fields (**name.firstName** and **name.lastName**) that you query and random encryption for six other fields (**name.otherNames**, **address**, **dob**, **phoneNumber**, **salary**, and **taxIdentifier**). All other fields are unencrypted.

The **address** field is an object, **dob** is an date, and **salary** is a double. All the other fields are all strings.

Create a Encryption Schema for this document using a single DEK. Remember to get your DEK UUID for the schema!

Encryption Schema Exercise - Server-side

Using the mongosh, modify the companyData.employee collection to include the Encryption Schema as a JSON schema validator. This does not require an encrypted client (an encrypted client cannot be used for this purpose). Remember the namespace is not included for the server-side JSON Schema Validator.

This will need to use:

```
// Use an unencrypted mongosh
use companyData;
db.runCommand({
  collMod: "employee",
  validator: {
    $jsonSchema: <SCHEMA_MAP_GOES_HERE>
  }
}
)
```

Encryption Schema Exercise

Using the code you created for manual encryption, or the mongosh, comment out all the encrypt methods and attempt to insert the unencrypted document into the database again.

What error do you get?

Inserting encrypted data

Compared to explicit encryption, implicit encryption is very easy.

Just use a regular insertOne/insert_one method to insert your documents. Bulk operations can be used as well.

Automatic Encryption Exercise

We will use an encrypted MongoClient this time, which requires the AutoEncryptionOptions object. As part of this object we need to provide the CA certificate and certificate PEM options to authenticate with the KMIP device. (Skeleton code for this exercise is in the automatic_encryption directory on the client)

```
auto_encryption = AutoEncryptionOpts(
 kms_provider, # variable for the key provider
 keyvault_namespace, // variable for the keyvault namespace string
 schema_map = schema_map,
 kms_tls_options = {
   "kmip": {
     "tlsCAFile": "/etc/pki/tls/certs/ca.cert",
     "tlsCertificateKeyFile": "/home/ec2-user/server.pem"
   }
 },
   crypt_shared_lib_required = True,
   mongocryptd_bypass_spawn = True,
   crypt_shared_lib_path = '/lib/mongo_crypt_v1.so'
)
```

Automatic Encryption Exercise

What is sent to the server if the following is executed with an encrypted MongoClient:

```
db.employee.updateOne(
  {"name": {"firstName": "Anggi", "lastName": "Breadtop"}},
  {"$set": {"taxIdentifier": "aswc7123_998"}},
  {"upsert": true}
)
```

What occurs if you run this command more than once? What will the **modifiedCount** be each time?

Reading and Writing Data

We can only query deterministically encrypted or unencrypted fields. Due to the nature of the encryption used, several commands and operators cannot be used when using a MongoDB client configured for automatic encryption. Only the following commands can be used with an automatic encryption MongoDB client:

- aggregate
- count
- delete
- distinct
- explain
- find
- findAndModify
- insert
- update

Supported Query Operators

The following operators can be used against a deterministically encrypted field with an automatic encryption configured MongoDB client:

- $eq
- $ne
- $in
- $nin
- $and
- $or
- $not
- $nor

Note that commands that run via **db.runCommand** and **db.adminCommand** cannot be executed in many cases when the MongoDB client is configured to use CSFLE.

Automatic Decryption Exercise

In the automatic_decryption directory, look at the code.

Attempt to query the salary field and see what happens.

Attempt to perform a range query the name.firstName field and see what happens.

Do you think you can use $out or $merge within the aggregation framework with encrypted data?

Use Cases

Key Per User

In some scenarios there is a requirement to have a DEK per user. This would require the application or a microservice to create the DEK for the user when the user is first registered or created. This means changing our custom role to allow the application user permissions to insert into the keyvault.

So how do we specify a key per user in our Encryption Schema?

We use JSON schema pointers which point to a field in the document we are encrypting. The field will contain the key alternative name of the key of interest, e.g. the user's DEK.

The key alternative name will be prefixed with a backslash.

Encryption Schema - Python

```
schema_map = {
  "<DATABASE>.<COLLECTION>": {
   "bsonType": "object",
   "properties": {
     "<FIELD NAME>": {
       "encrypt": {
         "bsonType": "string",
         "keyId": "/userID", # <- Field containing the DEK alternative name
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
       }
     },
   },
...
}
```

Note that this only works if every query to this collection includes the field containing the DEK Alternative name, which is **userID** in this case

Encryption Schema - Go

```
schemaMap = bson.M{
 "<DATABASE>.<COLLECTION>": bson.M{
   "bsonType": "object",
   "properties": bson.M{
     "<FIELD NAME>": bson.M{
       "encrypt": bson.M{
         "keyId": "/userID", // <- Field containing the DEK alternative name
         "algorithm": "AEAD_AES_256_CBC_HMAC_SHA_512-Random",
         "bsonType": "string",
       },
     },
...
}
```

Encryption Schema - Java

```
String dekId = "<DEK_UUID>";
Document jsonSchema = new Document().append("bsonType", "object").append("encryptMetadata",
       new Document().append("keyId", "/userID") // <- Field containing the DEK alternative name
           .append("algorithm", "AEAD_AES_256_CBC_HMAC_SHA_512-Random"))
   .append("properties", new Document()
       .append("<FIELD NAME>", new Document().append("encrypt", new Document()
           .append("bsonType", "string")
       ....);
HashMap<String, BsonDocument> schemaMap = new HashMap<String, BsonDocument>();
schemaMap.put("<DATABASE>.<COLLECTION>", BsonDocument.parse(jsonSchema.toJson()));
```

Key Per User

One limitation to note is that when using JSON schema pointers you can only use random encryption for those particular fields using the pointer, not deterministic, therefore you cannot query the fields encrypted with a pointer.

Imagine that each employee document in the "companyData.employee" collection has the encrypted fields encrypted by using a pointer, therefore all fields must be encrypted using random encryption, we would not be able to do anything, we could not find a simple document.

So how do we overcome this?

We can either

1- Use manual encryption with deterministic algorithm for the queryable fields (so we do not need these fields in the schema), or
2- Use a common DEK for the fields we need to query.
3- There is also the possibility that the field that is searched does not actually need to be encrypted. This is very much up to the customer's requirements.

Key Per User - Option 1

The issue with option 1 is that we need to know what the user's DEK UUID is, therefore we have to be able to get this. We can retrieve the DEK alternative name by querying on the keyAtlName field, but we need to know what the alternative name is in the first place. If a customer logs in with their customer number, then this could be the keyAltName.

What happens if the customer forgets their customerID? How do you find the DEK? You can't because their customer document is encrypted with their own personal DEK meaning we cannot even query for their name to find their customer document without knowing the DEK. This is a risk that must be understood before using this option.

Just remember, you need to know the unique DEK UUID of the customer to be able to find (if the name fields are used for querying) their document(s).

Key Per User - Option 2

For option 2 we would have a common DEK used to encrypt the name.firstName and name.lastName fields. In normal operations we do not query on the names because we use the customer ID.

In the case where the customer cannot remember their ID we would then use the name.firstName and name.lastName to query for the customer to retrieve the customer ID. This means customer support or the customer could retrieve their customer ID (which is not encrypted) via a name search, which would lead to their DEK keyAltName.

Whether the use of a common DEK for the **name.firstName** and **name.lastName** may be considered as a breach of privacy for right-to-forget purposes, this must be assessed before implementing.

Remember each document would be decrypted because of automatic decryption.

Key Per User - Option 2

For example, a search for John Smith, without the customer ID, will return to the application an array of user documents that contain John Smith as the name.

Because all the fields in the documents are automatically decrypted, we could identify the “correct” user by asking several other questions like address, phone number etc to pinpoint the correct user and using answer to security questions to avoid bad actor from stealing users information.

The application itself would need to search the array of decrypted documents to determine which document was the correct one from the further information we gather from the customer. We could not query for these fields because they are encrypted with random encryption, which cannot be queried.

Key Per User - Option 3

If encryption is not required for the query fields then this is an easy situation, but the combination of **name.firstName** and **name.lastName** may be considered as a breach of privacy for right-to-forget purposes, this must be assessed before implementing this option.

Create & Delete DEK

This uses the ClientEncryption class.

The method to create a DEK is:

```
<ClientEncrytion_Instance>.create_data_key(key_provider, master_key_id,
key_alt_name)
```

The method to delete a DEK is:

```
<ClientEncrytion_Instance>.delete_key(DEK_UUID)
```

Or a via normal MongoDB Client:

```
<KEY_VAULT_NAMESPACE>.deleteOne({"_id": UUID(DEK_UUID)})
```

Key Per User - Exercise

First we must modify our custom encryptClient role to allow insert and remove privileges on our key vault (__encryption.__keyVault). Use the mongosh to perform this task using db.grantPrivilegesToRole() method.

```
// use a normal (unencrypted) mongosh
use admin;
db.grantPrivilegesToRole(
  "cryptoClient",
  [
    {
      "resource": {"db": "__encryption", "collection": "__keyVault"},
      "actions": [ "find", "insert", "remove" ]
    }
  ]
)
```

Key Per User - Option 1 - Exercise

Using the code in the use_case_1_create directory as a template, perform the following enhancements:

- Create a new DEK for every new employee and include the DEK alternative name in the employee's document. Retrieve the DEK's UUID if it already exists.
- Change your Encryption Schema to use JSON pointers for the randomly encrypted fields that uses the employee's DEK
- Use manual encryption to encrypt the name.firstName and name.lastName fields with the employee's DEK
- Insert the document
- Query for the document you inserted using name.firstName and name.lastName
- Print the decrypted returned document

How will the driver retrieve the DEK if you do not know the keyAltNames or the _id of the DEK for the specific customer?

Key Per User - Option 1 - Exercise

```
Enterprise rs0 [primary] companyData> db.getSiblingDB("__encryption").getCollection("__keyVault").find({"keyAltNames":"53707"})
[
  {
    _id: new UUID("dd581f58-75ba-42bc-bbae-ed68ea691daf"),      << You cannot retrieve this if you do not know 53707, which mean no dek_UUID
    keyAltNames: [ '53707' ],
    keyMaterial: Binary(Buffer.from("605a2a125761c0c3af23e485818075d239.....d7b0904", "hex"), 0),
    creationDate: ISODate("2023-02-09T06:01:13.116Z"),
    updateDate: ISODate("2023-02-09T06:01:13.116Z"),
    status: 0,
    masterKey: { provider: 'kmip', keyId: '4' }
  }
]
```

Our code:

```
firstName = client_encryption.encrypt("Matt", Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic, dek_UUID)
lastName = client_encryption.encrypt("VBLover", Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic, dek_UUID)
doc = client.find_one({"name.firstName": firstName, "name.lastName": lastName})
```

Because we cannot get the UUID of our DEK to perform the manual encryption to use with our query, we cannot encrypt our query variables, and therefore we cannot find our document

Key Per User - Option 2 - Exercise

Using the code in the use_case_2_create directory as a template, perform the following enhancements:

- Create a common DEK to use to encrypt name.firstName and name.lastName of all documents. You can simply use the one created before.
- Create a new DEK for every new employee and include the DEK alternative name in the employee's document. Retrieve the DEK's UUID if it already exists.
- Change your Encryption Schema to use JSON pointers for the randomly encrypted fields that uses the employee's DEK and use your first DEK (dataKey1) for the name.firstName and name.lastName
- Insert the document
- Query for the document you inserted using name.firstName and name.lastName
- Print the decrypted returned document

Key Per User - Encryption Schemas

Notice that our server-side JSON Schema Validator did not error even though we are using different DEKs than what is declared in the server-side JSON Schema Validator? Remember that when the server-side checks, it only checks the fields are BSON binary subtype 6, not which algorithm or DEK is used.

If the client did not provide the Encryption Schema the server-side version would have been used and would have not satisfied our use case, therefore you should maintain the Encryption Schemas and ensure they align.

Key Per User - Right to Forget

A continuation of this use case with where there user has the right to forget. Because each user has their own DEK we can delete the DEK. The data remains in the database, but it cannot be decrypted because the DEK has been deleted. Customers must determine if this method of crypto-shredding satisfies their requirements for the Right to Forget.

In this case the application would need to be able to delete keys.

We will once again need to change our custom role to include the delete privilege on our key vault.

We can then use either ClientEncryption.**delete_key** method or simply **delete_one** method, in the format of:

```
<ClientEncrytion_Instance>.delete_key(DEK_UUID)
```

Key Per User - Exercise

Using the code in the use_case_delete directory as a template, perform the following enhancements:

- Delete the employee's DEK immediately after printing the document post-insertion
- Perform the query again and see if you can retrieve the document
- Insert a 60 sleep after this and then attempt the query again.
- Attempt to retrieve the document again
- Handle errors appropriately

Key Per User - Right to Forget

Remember **libmongocrypt** caches the unencrypted DEKs for 60 seconds, therefore if you delete the DEK and immediately query again you will most likely be able to retrieve the encrypted documents. There is no method to change this caching period.

If the MongoDB driver cannot retrieve the DEK it will throw an exception: you must handle this.

Key Per User - Right to Forget

If we encrypt the field we query on with a customer's or employee's DEK we have to know what DEK to use in advance so we can query the field.

This becomes impractical when we are trying to satisfy requirements where a customer provides their name and not their customer number, somehow you have to resolve their name to their unique identifier. This is a well know issue of the right-to-forget.

Further to this, if we delete a DEK that DEK will be in our backups.
How do we manage this?
This is another well know issue where regulations do not align with what the real world can achieve. Selectively removing portions of backups is dangerous, if it is even possible.

Key Rotation

Key Rotation

We can rotate the CMK to suit a customer's requirements. This means we will use a new CMK to re-wrap (aka re-encrypt) the DEKs. The actual DEKs themselves do not change, they are only re-wrapped with a different encryption.

We rotate the CMK via the **rewrapManyDataKey** method of **ClientEncryption** object. We provide this method with a filter to determine which DEKs to re-wrap, plus an object containing the new key provider (even if it is the same provider as our current CMK) and the CMK details.

Because we can provide the new key provider, we can move to a different key provider altogether, which is referred to as Key Migration, such as from KMIP to AWS.

We always recommend taking a full backup before rotating CMK.

ClientEncrypt - Mongosh

```
// start mongosh with `mongosh --nodb`
const provider = {
 "kmip": { // <-- KMS provider name
    "endpoint": "csfle-kmip-<PETNAME>.mdbtraining.net"
 }
}

const tlsOptions = {
  kmip: {
    tlsCAFile: "/etc/pki/tls/certs/ca.cert",
    tlsCertificateKeyFile: "/home/ec2-user/server.pem"
  }
}

const autoEncryptionOpts = {
 kmsProviders : provider,
 schemaMap: {}, //no Encryption Schema
 keyVaultNamespace: "__encryption.__keyVault",
 tlsOptions: tlsOptions
}
```

Rotate CMK - Mongosh

```
encryptedClient =
Mongo("mongodb://mongoadmin:passwordone@csfle-mongodb-<PETNAME>.mdbtraining.net:27017/?replicaSe
t=rs0&tls=true&tlsCAFile=%2Fetc%2Fpki%2Ftls%2Fcerts%2Fca.cert", autoEncryptionOpts)

keyVault = encryptedClient.getKeyVault()

keyVault.rewrapManyDataKey(
 {
    "masterKey.keyId": "<ID_OF_CMK>" // <-- example filter
 },
 {
    "provider": "kmip", // <-- new key provider name
    "masterKey": { // <-- CMK info (specific to KMIP in this case)
      "keyId": "<ID_OF_KEY>",
      "endpoint": "<KMIP_ENDPOINT>"
    }
 }
)
```

Rotate CMK - Exercise

Create a new CMK by executing:

python3 ~/kmip/new_cmk.py

Do a find() on the __encryption.__keyVault collection and determine the CMK keyId of the current DEKs.

Using the mongosh, create a ClientEncryption instance and rewrap all the DEKs with a new CMK that is in the same KMIP as the old CMK. Run the following to find the newest CMK ID:

python3 ~/kmip/show_cmks.py

Once completed, do a find() on the __encryption.__keyVault collection and determine if all DEKs have been rewrapped with the new CMK

Rotate CMK - Exercise Solution

Create a new CMK by executing:
python3 ~/kmip/new_cmk.py << TO CREATE
python3 ~/kmip/show_cmks.py << TO SHOW

```
var uri =
"mongodb://mongoadmin:<PASSWORD>@csfle-mongodb-<PETNAME>.mdbtraining.net:27017/?replicaSet=rs0&tls=true&tlsCAFile=%2Fetc%2Fpki%2Ftls%2Fcerts%2Fca.cert";

const keyVaultNamespace = "__encryption.__keyVault";

const kmsProviders = {kmip: {endpoint: "csfle-kmip-<PETNAME>.mdbtraining.net"}};

const tlsOptions = {
 kmip: {
   tlsCAFile: "/etc/pki/tls/certs/ca.cert",
   tlsCertificateKeyFile: "/home/ec2-user/server.pem"}}

var autoEncryptionOpts = {
   keyVaultNamespace: keyVaultNamespace,
   kmsProviders: kmsProviders,
   tlsOptions: tlsOptions
 }
encryptedClient = Mongo(uri, autoEncryptionOpts);
let keyVault = encryptedClient.getKeyVault()

keyVault.rewrapManyDataKey(
  {},{ // <- Note the filter is empty, which means all DEKs will be rewrapped
   "provider":"kmip", // <-- new key provider name
   "masterKey": { // <-- CMK info (specific to KMIP in this case)
     "keyId": "2",
     "endpoint": "csfle-kmip-star-marmot.mdbtraining.net"
   }})
```

Best Practises

Best Practices - CSFLE and Basic Cyber

Use **crypt_shared** if using MongoDB 4.4+.

If using mongocryptd, start in long-running daemon mode when the VM or container starts, and ensure the application encryptionClient configures autospawn to false.

The application user should only have encrypt and decrypt permissions for the Key Provider.

The application user should only have find access to DEKs, unless a right-to-forget or DEK creation requirement exists.

Always use a server-side schema validator.

Using private-peering between client and MongoDB (this is just good practice).

Take a full backup before rotating the CMK.

Audit access to CMK regularly

Best Practises - CSFLE and Basic Cyber

CSFLE is not a tool to enforce access control to fields, it is a tool to protect data. The MongoDB Client has access to all DEKs (if the application user has this permission) and therefore can decrypt any data.

Access control to access fields should be performed via your application itself or via views.

We recommend also using Encryption At Rest.

Conclusion

We have seen the features of CSFLE, plus some of the limitations.

We have used manual and automatic encryption and decryption techniques.

We have looked at some use cases.

We have rotated the CMK and rewrapped the DEKs with the new CMK.

Remember we can only query on fields that have been deterministically encrypted and that when using JSON pointers in our Encryption Schemas we can only use random encryption.

Appendix for Java users- maven

Maven is a build tool for java. It's already installed on your client (csfle-3) instances and has been used to download the needed libraries. By default you compile a project by entering at the command line:

```
mvn compile
```

A configuration file - pom.xml by default - is used to define builds. For each exercise there's a separate pom file you can specify to use the right source directory. The command for building the manual encryption exercise becomes:

```
maven -f pom_manual_encrypt.xml compile
```

The library files have been copied to `~/lib and` all class files are compiled to the directory `~/target/classes` so that the command for running the exercise never changes:

```
java -cp "./target/classes:./lib/*" App
```

Appendix for Java users - Reactive Streams

Reactive streams are a specification and associated Java API for asynchronous, non-blocking handling of streams of data with back-pressure.

This allows parallel use of hardware resources (e.g. CPU) while ensuring the stream destination does not get overwhelmed by buffering arbitrarily large amounts of data.

`See https://www.reactive-streams.org/`

The MongoDB Reactive Streams driver

The MongoDB asynchronous driver for Java implements the reactive streams interfaces. The CSFLE course uses this driver.

This means that MongoDB client CRUD operations return a reactive streams `Publisher<T>` instance. The `Publisher<T>` will have a `.subscribe(Subscriber<T> s)` method. When the publisher receives data it will pass it to any subscribed subscribers to be handled. `ClientEncryption` operations also use the reactive streams pattern.

See https://www.mongodb.com/docs/drivers/reactive-streams/

At the end of the `App.java` files you'll find a collection of classes implementing the `Subscriber<T>` interface for your convenience.

Java Users - MongoCollection - find

```
// Synchronous
Document encryptedResult = encryptedColl.find(eq("name.first_name", encryptedName));

// Asynchronous
ObservableSubscriber<Document> docSubscriber = new OperationSubscriber<Document>();
encryptedColl.find(eq("name.first_name", encryptedName)).subscribe(docSubscriber);
Document encryptedResult = docSubscriber.first();
```

Java Users - ClientEncryption - decrypt

```
// Synchronous
BsonValue decryptedValue = clientEncryption.decrypt(encryptedValue, encryptionOpts);
System.out.println(decryptedValue.toString();

// Asynchronous
ObservableSubscriber<BsonValue> sub = new ConsumerSubscriber<BsonValue>(
      decryptedValue -> System.out.println(decryptedValue.toString());
);
clientEncryption.decrypt(encryptedValue, encryptionOpts).subscribe(sub);
sub.await();
```

Java users- maven

Maven is a build tool for java. It's already installed on your client (csfle-3) instances and has been used to download the needed libraries. By default you compile a project by entering at the command line:

```
mvn compile
```

A configuration file - pom.xml by default - is used to define builds. For each exercise there's a separate pom file you can specify to use the right source directory. The command for building the manual encryption exercise becomes:

```
maven -f pom_manual_encrypt.xml compile
```

The library files have been copied to `~/lib and` all class files are compiled to the directory `~/target/classes` so that the command for running the exercise never changes:

```
java -cp "./target/classes:./lib/*" App
```

Appendix for Golang users

Initially you will need to perform:

```
source ~/.bash_profile
```

Each exercise directory will require you to perform the following:

```
go init mod csfle
go get go.mongodb.org/mongo-driver/bson
go get go.mongodb.org/mongo-driver/bson/primitive
go get go.mongodb.org/mongo-driver/mongo
go get go.mongodb.org/mongo-driver/mongo/options
go get github.com/goombaio/namegenerator
```

To compile or run you will need to perform the following:

```
go build -tags cse main.go
go run -tags cse main.go
```