Data Encryption in CM

Why do we need Transparent Encryption in HDFS?

When we want to encrypt only selected files/directories in HDFS to save on overhead and protect performance – now this is possible with HDFS Transparent Data Encryption (TDE).

HDFS TDE allows users to take advantage of HDFS native data encryption without any application code changes.

Once an HDFS admin sets up encryption, HDFS takes care of the actual encryption/decryption without the end-user having to manually encrypt/decrypt a file.

Building blocks of TDE (Transparent Data encryption)

1. **Encryption Zone (EZ):** An HDFS admin creates an encryption zone and links it to an empty HDFS directory and an encryption key. Any files put in the directory are automatically encrypted by HDFS.

2. **Key Management Server (KMS):** KMS is responsible for storing encryption key. KMS provides a REST API and access control on keys stored in the KMS.

3. **Key Provider API:** The Key Provider API is the glue used by HDFS Name Node and Client to connect with the Key Management Server.

Accessing data within an encryption zone

EZ Key – Encrypted zone key

DEK – Data encrypted keys

EDED – Encrypted DEK

- When creating a new file in an encryption zone, the NameNode asks the KMS to generate a new EDEK encrypted with the encryption zone’s key. The EDEK is then stored persistently as part of the file’s metadata on the NameNode.

When reading a file within an encryption zone, the NameNode provides the client with the file’s EDEK and the encryption zone key version used to encrypt the EDEK. The client then asks the KMS to decrypt the EDEK, which involves checking that the client has permission to access the encryption zone key version. Assuming that is successful, the client uses the DEK to decrypt the file’s contents.

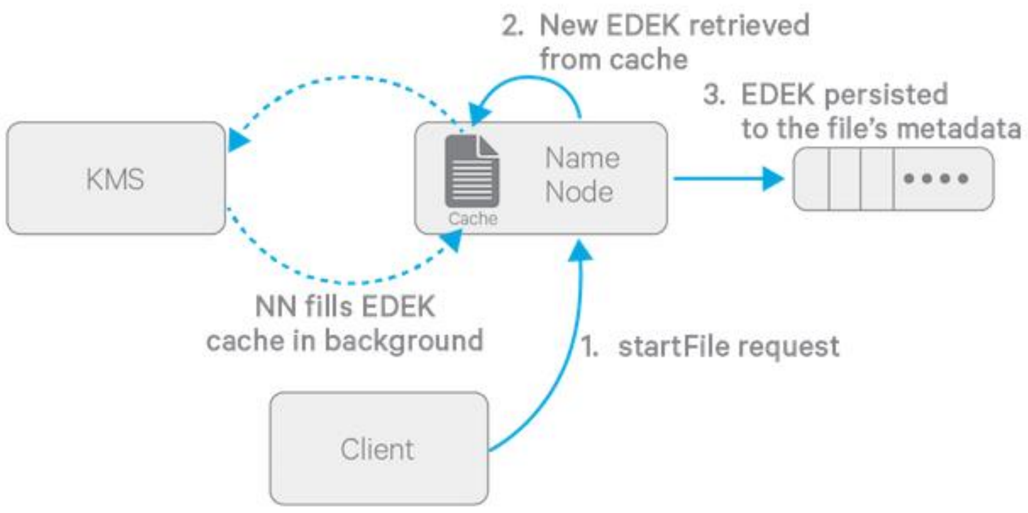
- All of the above steps for the read and write path happen automatically through interactions between the DFSClient, the NameNode, and the KMS.

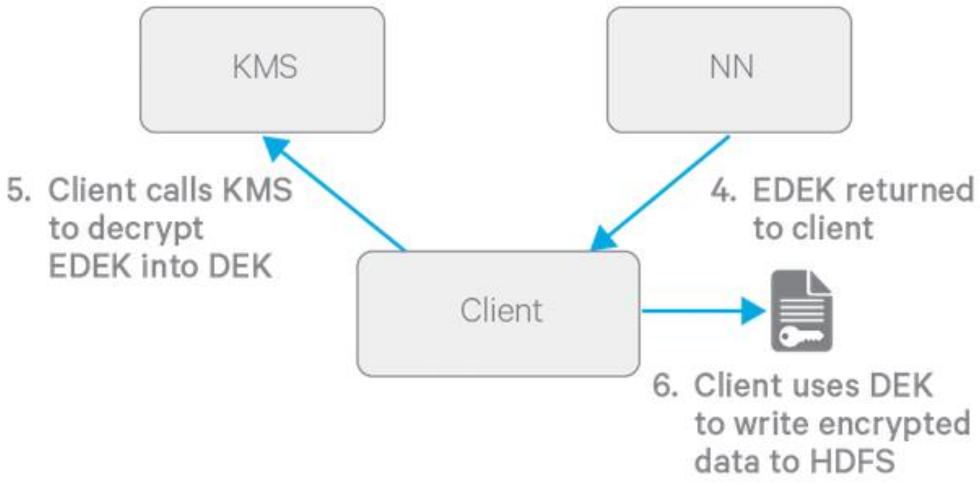
- Access to encrypted file data and metadata is controlled by normal HDFS filesystem permissions. This means that if HDFS is compromised (for example, by gaining unauthorized access to an HDFS superuser account), a malicious user only gains access to ciphertext and encrypted keys. However, since access to encryption zone keys is controlled by a separate set of permissions on the KMS and key store, this does not pose a security threat.

Use Cases of TDE

- Data encryption is required by a number of different government, financial, and regulatory entities. For example, the health-care industry has HIPAA regulations, the card payment industry has PCI DSS regulations, and the US government has FISMA regulations. Having transparent encryption built into HDFS makes it easier for organizations to comply with these regulations.

- Encryption can also be performed at the application-level, but by integrating it into HDFS, existing applications can operate on encrypted data without changes. This integrated architecture implies stronger encrypted file semantics and better coordination with other HDFS functions.


2. New EDEK retrieved from cache
3. EDEK persisted to the file's metadata
KMS
Cache
Name Node
NN fills EDEK cache in background
1. startFile request
Client


KMS
NN
5. Client calls KMS to decrypt EDEK into DEK
4. EDEK returned to client
Client
6. Client uses DEK to write encrypted data to HDFS