PARQUET-1854: Properties-driven Interface to Parquet Encryption Management

Contributors (chronological order): Ryan Blue (Netflix), Gidon Gershinsky (Apple), Xinli Shang (Uber), Maya Anderson (IBM), Gabor Szadovszky (Cloudera), Tham Ha (Emotiv)

Editor: Gidon Gershinsky (ggershinsky@apple.com)

Overview

Parquet encryption can be activated via configuration properties (such as Hive table properties, Hadoop configuration).

The “Examples” section below shows a couple of Spark samples that demonstrate the properties-driven encryption interface.

The implementation of the properties-driven encryption is heavily based on the Key Management Tools (PARQUET-1373). Since these tools are not specific to the properties-driven interface, and have a wider applicability, their design is described in a separate document.

Technical approach

Hadoop configuration

In parquet-mr, Hadoop configuration is passed to parquet writers and readers.
The following Hadoop properties allow to activate the properties-driven crypto factory, that in turn activates and configures encryption (file writing) and decryption (file reading):

Crypto factory

`“parquet.crypto.factory.class”` - set to
`“org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory”`

File writing

Mandatory properties

`"parquet.encryption.column.keys"`: list of columns to encrypt, with master key IDs.
(in the form of “<masterKeyID>:<colName>,<colName>;<masterKeyID>:<colName>...”)

`"parquet.encryption.footer.key"`: master key ID for footer encryption/signing - protects privacy and/or integrity of Parquet metadata

`"parquet.encryption.kms.client.class"`: name of class implementing KmsClient interface (see Key Management Tools). KMS stands for “key management service”.

Optional properties

`"parquet.encryption.algorithm"`: by default, “AES_GCM_V1” is used. This property can be used to switch to “AES_GCM_CTR_V1”

`"parquet.encryption.plaintext.footer"`: by default, files are written with an encrypted footer. This property can be used to write files with a plaintext footer (set to `“true”`). A footer key is still required to sign/tamper-proof the Parquet metadata (footer).

See Key Management Tools for detailed explanation of the following properties:

`"parquet.encryption.key.access.token"`: authorization token that will be passed to KMS (if KMS requires tokens)
`"parquet.encryption.kms.instance.id"`: ID of the KMS instance that will be used for encryption (if multiple KMS instances are available)

`"parquet.encryption.kms.instance.url"`: URL of the KMS instance

`"parquet.encryption.double.wrapping"`: by default, true - data encryption keys (DEKs) are encrypted with key encryption keys (KEKs), which in turn are encrypted with master keys. If set to false, DEKs are directly encrypted with master keys, KEKs are not used.

`"parquet.encryption.cache.lifetime.seconds"`: lifetime of cached entities (key encryption keys, local wrapping keys, KMS client objects). The default is 600 (10 minutes).

`"parquet.encryption.key.material.store.internally"`: by default, true - the key material is stored inside Parquet file footers - this doesn’t produce additional files. If set to false, key

material is stored in separate files in the same folder - this enables key re-wrapping for immutable Parquet files.

`"parquet.encryption.data.key.length.bits"`: length of data encryption keys (DEKs), randomly generated by the key management tools. Can be 128, 192 or 256 bits. The default is 128.

`"parquet.encryption.kek.length.bits"`: length of key encryption keys (KEKs), randomly generated by the key management tools. Can be 128, 192 or 256 bits. The default is 128.

File reading

Requires less properties, since most of them are stored in the file metadata.
The property explanations are identical to the writer-side counterparts.

Mandatory properties

`"parquet.encryption.kms.client.class"`

Optional properties

`"parquet.encryption.key.access.token"`

`"parquet.encryption.cache.lifetime.seconds"`

KMS migration

Files are encrypted using a certain KMS instance/URL. The KMS details are written in files' key material / metadata. Sometimes, enterprise use cases require migration of the master keys to another KMS instance (for high availability / disaster recovery / data migration to another geolocation). To support these requirements, Parquet key management tools provide the readers with KMS parameters that override the values stored in the file key material:

`"parquet.encryption.kms.instance.id"`: ID of the KMS instance that will be used for decryption

`"parquet.encryption.kms.instance.url"`: URL of the KMS instance

Hive Metastore properties

Hive Metastore allows users to set serde properties that are converted to Hadoop parameters and passed to Parquet.

```
CREATE TABLE my_table(name STRING, credit_card STRING)
 ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
  WITH SERDEPROPERTIES (
     'parquet.encryption.column.keys'='k2: credit_card',
     'parquet.encryption.footer.key'='k1')
 STORED AS parquet
```

Examples

Clone the Spark repo and build a runnable distribution (with -pHive option).

A proper demonstration of properties-driven encryption requires deployment of a real KMS server. We start with a Hello World example, that uses a simple “in-memory KMS” client which doesn’t need a server. The second example requires a running Vault server. Hashicorp Vault is an open source KMS.

Both KMS clients are available in this jar - download and add it to the Spark `jars` folder.

Note: Vault KMS Client is provided only as a sample for KMS Client developers. It should not be used in production, and is not guaranteed to work if Vault API changes in the future. “In-memory KMS” does not demonstrate interaction with an external KMS, and therefore should not be even considered as a sample for KMS Client developers.

Both examples presume the `sampleDF` dataframe has a column named “col0” - which will be encrypted.

In all examples, the crypto factory should be set to the `PropertiesDrivenCryptoFactory` :

```
sc.hadoopConfiguration.set("parquet.crypto.factory.class" ,
                   "org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory")
```

“Hello World”

```
sc.hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                           "org.apache.parquet.crypto.keytools.mocks.InMemoryKMS")

// Explicit master keys, base64 (KMS-specific parameter, for InMemoryKMS)
```

```
sc.hadoopConfiguration.set("parquet.encryption.key.list" ,
                        "k1:AAECAwQFBgcICQoLDA0ODw== ,  k2:AAECAAECAAECAAECAAECAA==")


// Write/encrypt
sampleDF.write.
   option("parquet.encryption.footer.key" , "k1").
   option("parquet.encryption.column.keys" , "k2:col0").
parquet("/path/to/table.parquet.encrypted")


// Read/decrypt
val df2 = spark.read.parquet("/path/to/table.parquet.encrypted")
```

Hashicorp Vault KMS

```
sc.hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                        "org.apache.parquet.crypto.keytools.samples.VaultClient")

// Vault server is set up with
// 1) a Transit secrets engine enabled,
// 2) two master keys in the Transit engine, named “k1” and “k2” (non-exportable keys)
// 3) a token, and a policy allowing the token bearer to use these keys


// Write/encrypt
sc.hadoopConfiguration.set("parquet.encryption.key.access.token" , "<vault token>")
sc.hadoopConfiguration.set("parquet.encryption.kms.instance.url" , "<vault server
url>")

sampleDF.write.
   option("parquet.encryption.footer.key" , "k1").
   option("parquet.encryption.column.keys" , "k2:col0").
parquet("/path/to/table.parquet.encrypted")


// Read/decrypt
sc.hadoopConfiguration.set("parquet.encryption.key.access.token" , "<vault token>")
val df2 = spark.read.parquet("/path/to/table.parquet.encrypted")
```