CONTENTS

Introduction

Apache Community have provided nice option for beeline users added in HIVE-7175[1]. From now on it is possible to not enter password when authorization takes place but create a file with password and pass it as an option to beeline command. But password file is stored “as is” which is not great due to security reasons. The main point of this documentation is to create simple way of password file encryption using hive.

First of all let’s list authentication options that are currently provided alongside with that one that will be described in this document:

1. Login/Password pair - providing login password pair while authenticating;
2. Login/PathToPasswordFile - providing login and path to password file where password for authentication is stored.
3. The matter of this document - Login/PathToEncryptedPasswordFile - providing login and path to encrypted password file

See the general representation of authentication through Beeline CLI on Figure #1.

The feature described in this document may be implemented in two ways:

1. Store password in Java Key Store file using CredentialProvider. We can create keystore using hadoop command (`hadoop credential create <predefined secret key property name> -provider <path to keystore>`) and write password that will be used to connect through Beeline CLI.

2. Encrypt/decrypt password file using hive features, and store secret key for encryption/decryption in some way.

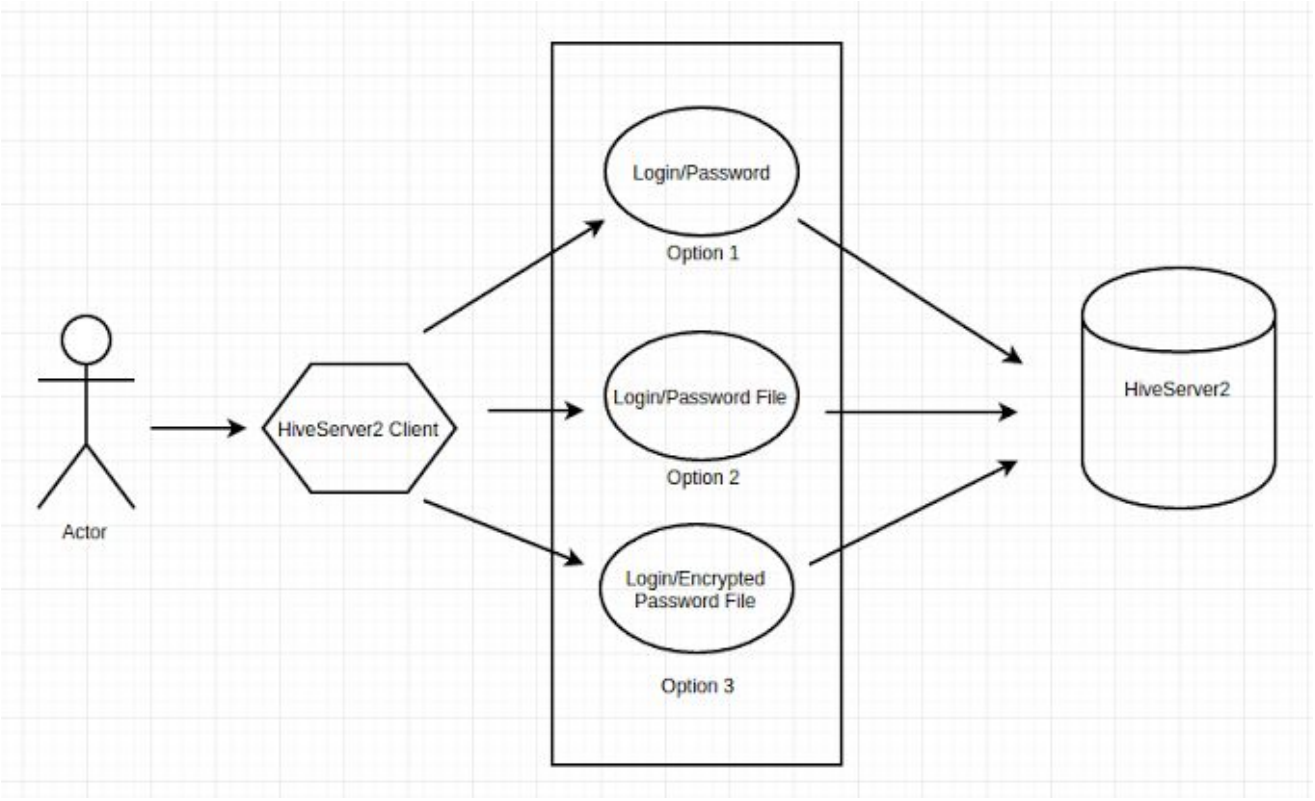

Figure #1 Types of authorization using Beeline CLI

Password file encryption using CredentialProvider API

Credential Provider API

The CredentialProvider API is an SPI framework for plugging in extensible credential providers. Credential providers are used to separate the use of sensitive tokens, secrets and passwords from the details of their storage and management. The ability to choose various storage mechanisms for protecting these credentials allows us to keep such sensitive assets out of clear text, away from prying eyes and potentially to be managed by third party solution[2].

Encryption method

Triple DES is another mode of DES operation. It takes three 64-bit keys, for an overall key length of 192 bits. In Stealth, you simply type in the entire 192-bit (24 character) key rather than entering each of the three keys individually. The Triple DES DLL then breaks the user-provided key into three subkeys, padding the keys if necessary so they are each 64 bits long. The procedure for encryption is exactly the same as regular DES, but it is repeated three times, hence the name Triple DES. The data is encrypted with the first key, decrypted with the second key, and finally encrypted again with the third key.

Triple DES runs three times slower than DES, but is much more secure if used properly. The procedure for decrypting something is the same as the procedure for encryption, except it is executed in reverse. Like DES, data is encrypted and decrypted in 64-bit chunks. Although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored, so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits. This means that the effective key strength for Triple DES is actually 168 bits because each of the three keys contains 8 parity bits that are not used during the encryption process[3].

Encryption implementation

CredentialProvider API is responsible for managing Java Key Stores where passwords are stored securely using Triple DES method provided by Java. To get the password from keystore Configuration.getPassword() method should be invoked.

Encryption of password file

To create encrypted password file we should hadoop credential command. Generally command looks like

```
hadoop credential create <predefined secret key property name>  -provider <path to keystore>
```

Keystores may be created on local machine as well as in distributed file system. Configuration property should be created to identify the password.

Beeline CLI authorization

Generally Beeline CLI connection command will look like:

hive --service beeline [-u <connection url>] [-n <username>] [-w <path to encrypted file>][-<key to identify that file is encrypted>]

Use cases

For all types of cluster security the steps for encrypted password file will be the same:

1. Create key storage. In case of using CredentialProvider we will need to pass secret key to store it after creation:

```
hadoop credential create <predefined secret key property name> -provider <path to keystore>
```

2. Authorize through Beeline CLI

```
hive --service beeline [-u <connection url>] [-n <username> ] [-w <path to encrypted file>] [-<key for identifying keystore path> <path to keystore>]
```

Password file encryption using secret key stored in separate file

Encryption method

Supported type of encryption is AES with different length of keys (128, 192, 256). AES is based on a design principle known as a substitution-permutation network, a combination of both substitution and permutation, and is fast in both software and hardware. Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification *per se* is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits[4].

Encryption implementation

Generally encryption and decryption steps will be performed through already implemented classes in Hive 2.0 GenericUDFAesDecrypt and GenericUDFAesEncrypt[5]. They are implementing encryption/decryption of String or byte array using secret defined by user.

Keystore implementation

Here will be provided description of several possible ways to store keys. General representation is shown on Figure #2:

1. **Java Key Store**

Java Keystore is a container for authorization certificates or public key certificates, and is often used by Java-based applications for encryption, authentication, and serving over HTTPS. Its entries are protected by a keystore password. A keystore entry is identified by an *alias*, and it consists of keys and certificates that form a trust chain[6].

1.1. Java KeyStore creation on local machine:

It is possible to create keystore locally using Java Keytool. Java Keytool is a key and certificate management tool that is used to manipulate Java Keystores, and is included with Java. This type of storing keys is not preferable due to complexity of Java KeyStore creation.

1.2 CredentialProvider

Another way to store keys with JavaKeystore is to use CredentialProvider. It is possible to create keystore locally or in distributed file system and write there any values that should be securely stored

2. **Key file**

Create file with secret key that will store secret key itself. The other possible way is to provide key/value pairs so the functional ability may be extended in future. **The main disadvantage is that we won't be able to strongly secure this files. The only possible way is to modify access permissions.**

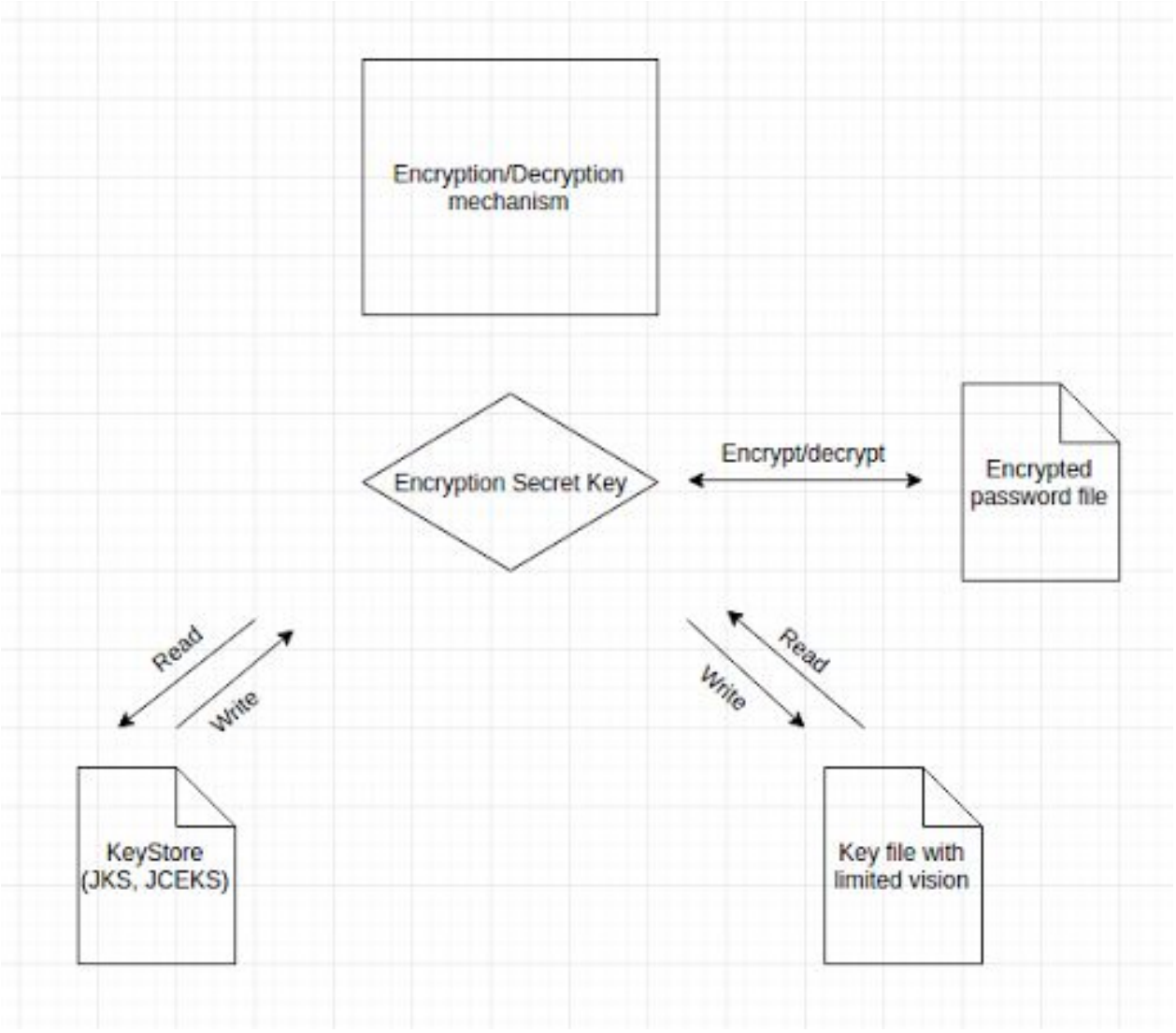

Figure #2 Possible options of storing secret key

Encryption of password file

Password encryption will be performed while executing hive with specific keys provided:

1. --encrypt <path to password file> - specify file where password is stored.
2. --key <secret key string>- provide secret key that will be used for file encryption.
3. --autogenerate - create secret key using hive built-in function.
4. --keystore <path to keystore file>

Generally command will look like:

```
hive [--encrypt <path to file>] [--key <secret key string> | --autogenerate] [--keystore <path to file>]
```

Beeline CLI authorization

Generally Beeline CLI connection command will look like:

```
hive --service beeline [-u <connection url>] [-n <username> ] [-w <path to encrypted file>] [-<key for identifying keystore path> <path to keystore>]
```

Use cases

For all types of cluster security the steps for encrypted password file will be almost the same:

1. Create password file

```
nano <path to password file>
```

2. Create key storage.

- In case of using CredentialProvider we will need to pass secret key to store it after creation:

```
hadoop credential create <predefined secret key property name> -provider <path to keystore>
```

- If key will be stored in regular file, the file will be created automatically in the next step.

3. Encrypt password file

```
hive [--encrypt <path to file>] [--key <secret key string> | --autogenerate] [--keystore <path to file>]
```

4. Authorize through Beeline CLI

```
hive --service beeline [-u <connection url>] [-n <username> ] [-w <path to encrypted file>] [-<key for identifying keystore path> <path to keystore>]
```

References

1. https://issues.apache.org/jira/browse/HIVE-7175
2. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/CredentialProviderAPI.html
3. https://www.vocal.com/cryptography/tdes/
4. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
5. https://issues.apache.org/jira/browse/HIVE-11593
6. https://www.digitalocean.com/community/tutorials/java-keytool-essentials-working-with-java-keystores