KIP-486: Support custom way to load KeyStore and TrustStore

Status

Current state: DISCARDED

Discussion thread: https://www.mail-archive.com/dev@kafka.apache.org/msg99126.html

Voting thread: https://www.mail-archive.com/dev@kafka.apache.org/msg100715.html

JIRA: KAFKA-8621

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Discard Reason

The discussion on this KIP lead us to create KIP-519: Make SSL context/engine configuration extensible which is in ACCEPTED state now and this particular KIP is not needed anymore.

Motivation

Current Kafka versions supports file based KeyStore and TrustStore via ssl.keystore.location and ssl.truststore.location configurations along with required passwords configurations.

This configuration requirement creates challenges for larger **Kafka deployment with following setup**,

- Having a company internal CA authority issuing the certificates
- Thousands of brokers
- 10x/100x number of Kafka Client boxes
- Requirement to use client auth
- Need to avoid storing key/trust store files on file system for stronger security
- Polyglot client base

The challenges are,

- Deploying the key/trust store files to the thousands of brokers
- Deploying the key/trust store files to 10x/100x Kafka Client boxes
- Keep all those key/trust store files and their passwords secure
- Operationally manage key rotations
- Lack of unified way to distribute KeyStore and TrustStores for different languages

Primarily the requirement of having the KeyStore and TrustStore locations on the file system manifests itself into many of the challenges listed above.

Assuming we have a custom Key Manager which can provide secure API to provision/access the keys we must find an alternative to the file based KeyStore and TrustStore.

The primary motivation here is to essentially provide an optional way to allow custom way to load KeyStore and TrustStore instead of relying on the file system. Of course, that needs to be accompanied by having an ability to load required passwords (for KeyStore, TrustStore and Keys) accordingly.

Public Interfaces

We will introduce an **optional** way to load KeyStore and TrustStore along with their required passwords as applicable.

This will be done via

1. Introducing two new configurations- ssl.keystore.loader and ssl.truststore.loader.
2. For each of the new configuration, we will have public interfaces as described below,

KeyStoreLoader

```
package org.apache.kafka.common.security.ssl;

import java.security.KeyStore;

public interface KeyStoreLoader {
        /**
         * This loads the keystore. The keystore password will be fetched by whatever mechanism the
implementation of this class chooses.
         * Example: It could use current ssl.keystore.password configuration if it chooses.
         * @return KeyStore object
         */
        public KeyStore load();

        /**
         * This returns the key's password.
         */
        public String getKeyPassword();

        /**
         * This method checks if the given keystore has been modified based on some criteria, typically last-
modified timestamp. The definition of 'modified' is left to the
         * implementation.
         * @return true - If the keystore was modified as defined by the implementation; false otherwise
         */
        public boolean modified();
}
```

TrustStoreLoader

```
package org.apache.kafka.common.security.ssl;

import java.security.KeyStore;

public interface TrustStoreLoader {

        /**
         * This loads the truststore. The truststore password will be fetched by whatever mechanism the
implementation of this class chooses.
         * Example: It could use current ssl.truststore.password configuration if it chooses.
         * @return KeyStore object
         */
        public KeyStore load();

        /**
         * This method checks if the given truststore has been modified based on some criteria, typically last-
modified timestamp. The definition of 'modified' is left to the
         * implementation.
         * @return true - If the truststore was modified as defined by the implementation; false otherwise
         */
        public boolean modified();
}
```

Why we do not specify key/trust store password as input method arguments in the interfaces?

We are not specifying the key/trust store passwords in the KeyStoreLoader/TrustStoreLoader load() method. This is because we want to avoid the dependency in the caller class to load the password. This implementation leaves it open to the Loader implementation to read required configuration or use other mechanism for fetching the password. Typically if you have a Key Manager solution you might be using some sort of 'auth-token' in order to access the Key Manager's API and might not require key/trust store password (you will still need password for unlocking the keys though).

Proposed Changes

Kafka Client library and Kafka Broker both uses SslEngineBuilder class to load KeyStore and TrustStore from the file based configurations.

1. As documented in public interfaces section, we will introduce two interfaces to allow pluggable implementation to provide key/trust stores loading
2. We will make changes to the SslEngineBuilder#createSSLContext() method to invoke the key/trust store loading from new ssl configurations we introduce.
    a. Pseudocode changes in the SslEngineBuilder#createSSLContext() looks like below

```
String kmfAlgorithm = this.kmfAlgorithm != null ? this.kmfAlgorithm : KeyManagerFactory.
getDefaultAlgorithm();
KeyManagerFactory kmf = KeyManagerFactory.getInstance(kmfAlgorithm);
KeyStore ks;
...
...
if ( keystore != null ) {
 // load keystore 'ks' in existing way
} else if ( 'ssl.keystore.loader' specified ) {
 // load keystore 'ks' by invoking the pluggable implementation class for the config
}
kmf.init(ks, ksPassword.toCharArray());
...
...
...
String tmfAlgorithm = this.tmfAlgorithm != null ? this.tmfAlgorithm : TrustManagerFactory.
getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
KeyStore ts = null;
...
if ( truststore != null ) {
 // load truststore 'ts' in existing way
} else if ( 'ssl.truststore.loader' specified ) {
 // load truststore 'ts' by invoking the pluggable implementation class for the config
}
...
tmf.init(ts);
...
```

3. We will make changes to the SslEngineBuilder#shouldBeRebuilt() method appropriately

Compatibility, Deprecation, and Migration Plan

- What impact (if any) will there be on existing users?

Existing users using file based key/trust stores are not going to be impacted at all.

- If we are changing behavior how will we phase out the older behavior?

Older behavior does not change so no need to phase it out.

- If we need special migration tools, describe them here.

No special migration tools needed.

- When will we remove the existing behavior?

We will keep the existing behavior and add optional new behavior.

Rejected Alternatives

Using existing ssl.provider config

We did experiment using ssl.provider config and we wrote a sample provider like below. However it didn't work for us since our provider does not have implementation for SSLContext.TLS/TLSv1.1/TLSv1.2 etc.

We must not have to add implementation for SSL context classes in our provider since we **only** intend to customize the TrustStoreManager in below example.

MyProvider

```
package providertest;

import java.security.Provider;

public class MyProvider extends Provider {

    private static final String name = "MyProvider";
    private static double version = 1.0d;
    private static String info = "Maulin's SSL Provider v"+version;

    public MyProvider() {
        super(name, version, info);
        this.put("TrustManagerFactory.PKIX", "providertest.MyTrustManagerFactory");
    }
}
```

Writing a Java security provider and registering it from JRE

Alternative to using ssl.provider configuration is to register the Java security provider in JRE's jre/lib/security/java.security file. This way we won't run into the limitation mentioned in the above rejected approach.

However there are following challenges,

1. We have to modify the java.security file on the system which creates the similar challenge as of hosting jks on local file system - meaning maintaining per box, deployment etc
2. Assuming modifying java.security file is not a challenge (See KIP-492) , we still have to write a Provider with **custom algorithm** for TrustManagerFactory and KeyManagerFactory
    a. When we write those factories implementation there is **no** easy way to re-use validation logic (example: OpenJDK TrustManagerImpl) done by existing Providers.
    b. The X509ExtendedTrustManager class is having all the methods **abstract** so we can't re-use any standard implementations easily
    c. We will end up **copying** the validation logic (example: OpenJDK TrustManagerImpl) which is "security domain" centric
    d. The validation logic deals with essentially three things as of now,
        i. client side cert checks
        ii. server side cert checks
        iii. certificate path validations
        iv. end point identification verification
    e. The above validation logic we should not have to deal with it in the first place for the purpose of **just loading keys/certs from a different source than the file based key/trust stores**.

NOTE: You can only realize the above challenges once you try to write the Provider with Trust/Key Manager factories. We would highly encourage you to try **writing** (using any other open-source library's provider as an example may not give you the idea) a provider to do this before you decide to comment on this approach.

One suggestion could be - **Why not use Java's inbuilt rails to "use any provider's implementation" for key/trust manager AND just plugin our own keys/certs?**

That is exactly what we are suggesting to do. Below is the example from our pseudo-code for using TrustManagerFactory.getInstance(). Same applies for KeyManagerFactory.

```
String tmfAlgorithm = this.tmfAlgorithm != null ? this.tmfAlgorithm : TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
KeyStore ks = < load trust store from either the local file or other source >
tmf.init(ks);
```

Other reason for rejecting this approach

Provider for "standard algorithms" are written in order to be re-used. If we just write a Provider tied to a specific way for loading Trust/Key stores defeats the purpose of the re-use of the Providers.

Provide a way to delegate SSLContext creation

We could create a new configuration like ssl.context.loader/ssl.context.initializer and use the implementation class to obtain the object of javax.net.ssl.SSLContext instead of using SslEngineBuilder#createSSLContext().

This is more work than actually needed and we looked at Keys/Secrets' Managers like Hashicorp's Vault as a sample integration and realized that the Vault API provides us way to get the Keys/Secrets/Certs but not the SSLContext object we need. This will be true for most Key Managers since it's primary responsibility is to manage keys/secrets/certs but **not** the SSLContext.

Also, if we do provide a way to create SSLContext in a custom way, we must still honor the "provider" value used in SslEngineBuilder#createSSLContext().

Overall, we didn't find enough justification to follow this path.

Generated the required SSL configuration values from the Key Manager API

If we have a Key Manager solution which provides APIs like Hashicorp's Vault we need to find a way to generate the required ssl configurations for Kafka (key/trust stores files, passwords etc) from the same.

We could,

- Build a mechanism to download the required keys/certs from the Key Manager API and create required key/trust store files
- Standardize the path to the key/trust stores files on the disk
- Protect the key/trust stores files & Kafka Broker/Client files with appropriate permissions
- Generate the required configurations for Kafka Client boxes and Kafka Brokers

... AND we will not need any customization we are talking about here.

However, this approach does not scale for managing deployments across many brokers and client boxes and it does not solve for any of the challenges mentioned in the "Motivation" section.

Hence this approach was rejected.