KIP-519: Make SSL context/engine configuration extensible

Status

Current state: ACCEPTED

Voting thread: https://www.mail-archive.com/dev@kafka.apache.org/msg104580.html

Discussion thread: https://www.mail-archive.com/dev@kafka.apache.org/msg101011.html

JIRA: ⚠ Unable to render Jira issues macro, execution error.

PR: https://github.com/apache/kafka/pull/8338

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

Motivation

Kafka adoption is growing day-by-day and it is used by all sizes of organizations serving varied technical and business domains like banking, entertainment, food/transportation logistics, financial technology to name a few. These organizations depending upon their size and domain may have more internal and external InfoSec/AppSec/Compliance standards they have to meet while using Kafka. SSL/TLS communication is very critical part of those standards. While Kafka supports SSL communication sometimes it becomes challenging to use out-of-the-box solution provided by Kafka for SSL /TLS given the organization's needs.

Currently Kafka provides several SSL related configurations that typically starts with 'ssl.' prefix. Those configurations focuses on ability to specify Keystore and Truststores parameters, supply custom java security Providers etc. However as mentioned before there is still room to allow more flexibility /extensibility to allow organizations to fully comply to their requirement without one-off customizations of Kafka.

According to JSSE Documentation **SSLContext and SSLEngine are the endpoint classes for the secure connection. Moreover, SSLEngine is created by SSLContext** and can be further customized according to the needs. Hence if Kafka provides a way to customize SSLContext and/or SSLEngine it would be a great lever and provide the ultimate extensibility for Kafka users.

This work is influenced by past KIPs (mentioned below) and is a fresh attempt to provide a conclusive solution. It also got more energy from discussion on KIP-486: Support custom way to load KeyStore and TrustStore.

Past KIPs,

KIP-76 Enable getting password from executable rather than passing as plaintext in config files

KIP-383: Pluggable interface for SSL Factory

This KIP proposes the work on top of what has been done already for SSL configuration like KIP-226 - Dynamic Broker Configuration and KIP-492: Add java security providers in Kafka Security config.

Our primary goal is: Instead of keep adding various ssl configurations to customize smaller portions of SSLContext/Engine, we should have a single configurable/pluggable SSLContext/Engine functionality.

Pluggable SSLContext or SSLEngine?

Java's Security Providers architecture already enables using custom SSLEngine by using the right JSSE compatible Provider like BouncyCastleJsseProvider. We could use any SSLEngine of the TLS implementations as far as we have JSSE compatible provider available. We can use Kafka's 'security.providers' configuration to use a custom provider.

Making javax.net.ssl.SSLContext setup pluggable provides flexibility for providing Key Material, Secure Random implementation and configure Key/Trust managers in a custom way. Example: Apache HttpComponents and Netty SslContextBuilder.

However, Kafka configures the SSLEngine for Client and Server both modes. Hence according to existing code it would be useful to make SslEngineBuilder pluggable. That will provide us a way to configure SSLContext object in a flexible way and at the same time will allow creation of SSLEngine with Client /Server mode.

Public Interfaces

New configuration

ssl.engine.factory.class - This configuration will take class of the below interface's type and will be used to create javax.net.ssl.SSLEngine object.

Default value will be as mentioned below.

```
    public static final String SSL_ENGINE_FACTORY_CLASS_CONFIG = "ssl.engine.factory.class";
    public static final String DEFAULT_SSL_ENGINE_FACTORY_CLASS = org.apache.kafka.common.security.ssl.
DefaultSslEngineFactory.class.getCanonicalName();
    public static final String SSL_ENGINE_FACTORY_CLASS_DOC = "The class of type org.apache.kafka.common.
security.auth.SslEngineFactory to provide SSLEngine objects. Default value is " +
DEFAULT_SSL_ENGINE_FACTORY_CLASS;
```

Interface for SslEngineFactory

Below is the interface suggested for this.

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

import org.apache.kafka.common.Configurable;

import javax.net.ssl.SSLEngine;
import java.io.Closeable;
import java.security.KeyStore;
import java.util.Map;
import java.util.Set;

/**
 * Plugin interface for allowing creation of SSLEngine object in a custom way.
 * Example: You want to use custom way to load your key material and trust material needed for SSLContext.
 * However, keep in mind that this is complementary to the existing Java Security Provider's mechanism and not
a competing
 * solution.
 */
public interface SslEngineFactory extends Configurable, Closeable {

    /**
     * Create a new SSLEngine object to be used by the client.
     *
     * @param peerHost               The peer host to use. This is used in client mode if endpoint validation
is enabled.
     * @param peerPort               The peer port to use. This is a hint and not used for validation.
     * @param endpointIdentification Endpoint identification algorithm for client mode.
     * @return The new SSLEngine.
     */
    SSLEngine createClientSslEngine(String peerHost, int peerPort, String endpointIdentification);

    /**
     * Create a new SSLEngine object to be used by the server.
     *
     * @param peerHost               The peer host to use. This is used in client mode if endpoint validation is enabled.
     * @param peerPort               The peer port to use. This is a hint and not used for validation.
     * @return The new SSLEngine.
     */
    SSLEngine createServerSslEngine(String peerHost, int peerPort);

         /**
     * Returns true if SSLEngine needs to be rebuilt. This method will be called when reconfiguration is triggered on
     * {@link org.apache.kafka.common.security.ssl.SslFactory}. Based on the <i>nextConfigs</i>, this method will
     * decide whether underlying SSLEngine object needs to be rebuilt. If this method returns true, the
     * {@link org.apache.kafka.common.security.ssl.SslFactory} will re-create instance of this object and run other
     * checks before deciding to use the new object for the <i>new incoming connection</i> requests.The existing connections
     * are not impacted by this and will not see any changes done as part of reconfiguration.
     *
     * <pre>
     *     Example: If the implementation depends on the file based key material it can check if the file is updated
     *     compared to the previous/last-loaded timestamp and return true.
     * </pre>
     *
     * @param nextConfigs       The configuration we want to use.
     * @return                  True only if the underlying SSLEngine object should be rebuilt.
     */
    boolean shouldBeRebuilt(Map<String, Object> nextConfigs);

    /**
     * Returns the names of configs that may be reconfigured.
     */
    Set<String> reconfigurableConfigs();

    /**
     * Returns keystore.
     * @return
     */
    KeyStore keystore();

    /**
     * Returns truststore.
     * @return
     */
    KeyStore truststore();
}
```

Proposed Changes

Currently SslFactory.java uses SslEngineBuilder.java. Instead of that we will modify SslFactory.java to load a class configured via the new configuration 'ssl.engine.factory.class' and delegate the SSLEngine creation call to the implementation.

We will also provide default implementation for the 'ssl.engine.factory.class' which will be used in absence of provided config.

Default implementation would be very similar to the current SslEngineBuilder.java.

Which classes will be deleted?

- SslEngineBuilder.java (functionality will be moved to DefaultSslEngineFactory.java)

Which classes will be added?

- SslEngineFactory.java Interface
- DefaultSslEngineFactory.java (mostly having code from existing SslEngineBuilder)

Which classes will be modified primarily?

- SslFactory.java

How does configs get to the implementation class?

The configuration of Map will be passed to the implementation class via the configure() method. See below example,

```
public DefaultSslEngineFactory implements SslEngineFactory {
...
...
        /* Default empty argument constructor */

        /* implement configure() method */
    @Override
    public void configure(Map<String, ?> configs) {
    ...
    }
...
...
}
```

These configuration will be passed from SslFactory to the implementation of the SslEngineFactory interface via reflection like below

```
public class SslFactory implement Reconfigurable {
...
...
     private SslEngineFactory instantiateSslEngineFactory(Map<String, Object> configs) {
        @SuppressWarnings("unchecked")
        Class<? extends SslEngineFactory> sslEngineFactoryClass =
                (Class<? extends SslEngineFactory>) configs.get(SslConfigs.SSL_ENGINE_FACTORY_CLASS_CONFIG);
        SslEngineFactory sslEngineFactory = Utils.newInstance(sslEngineFactoryClass);
        sslEngineFactory.configure(configs);
        this.sslEngineFactoryConfig = configs;
        return sslEngineFactory;
    }
...
}
```

Support for reconfiguration of custom configs

By custom configs we mean the configs used by the SslEngineFactory's implementation. Those configs does not have to be part of definition of Kafka configs since only the implementation class knows what are those. Kafka already supports custom configs so this should not be a new challenge.

Compatibility, Deprecation, and Migration Plan

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

There will be no impact on existing users who do not specify the new configuration.

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

No impact

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

None

- *When will we remove the existing behavior?*

Not applicable since old code behavior will be kept with default implementation of DefaultSslEngineFactory and modification to SslFactory class.

Rejected Alternatives

Make SSLEngine pluggable

As noted in the motivation section- Java's Security Providers architecture already enables using custom SSLEngine by using the right JSSE compatible Provider hence Kafka should not have to derive another way to make SSLEngine pluggable.

Making SslFactory the pluggable interface (KIP-383)

This is because currently SslFactory does certain validations which we want to keep separate and mandate those checks across any possible implementation of pluggable ssl context class. Also, once we start writing the reconfigurable classes we realize that we need two classes - 1) SslEngineFactory implementation and 2) Container of the factory implementation. We believe that keeping SslFactory as Reconfigurable object and help reconfigure the underlying SslEngineFactory will simplify the implementations of SslEngineFactory.

Also, we rejected to make SslEngineFactory extend the Reconfigurable interface due to following reason,

There will be good amount of state in the SslEngineFactory's implementation (as it will be similar to the current SslEngineBuilder class). We believe that making SSLContext creation and SSLEngine object's configuration pluggable is worth to allow SSL experts to write their own implementation having the SSL domain knowledge and keep them free of knowing much about Kafka's reconfigurability - example: Apache HttpComponents. We prefer SslFactory class to do what it is doing right now and keep the responsibility of re-creating underlying SslEngineFactory object based on the configurations specified by the SslContextFactory's implementation.

Creating builder for SSLContext

We could create a builder for SSLContext object and have a mechanism to configure SSLEngine object for client/server mode non-pluggable. However, creating a builder interface with options to build SSLContext will need to have method(s) to allow keys/trusted-certs. It will also require us to have 'key-password' as input for the keystore. In the current Kafka implementation it requires the password to be configured in the plaintext via 'ssl.key.password', 'ssl.keystore.password' and 'ssl.truststore.password'. If we need to customize how the password is loaded, due to security reasons, this approach will not work since some other mechanism for making password pluggable (See KIP-76 Enable getting password from executable rather than passing as plaintext in config files AND KIP-486: Support custom way to load KeyStore and TrustStore) need to be devised which will add more ssl related configurations to Kafka.