CEP-9: Make SSLContext creation pluggable

- Status
- Motivation
- Audience
- Goals
- Non-Goals
- Proposed Changes
- New or Changed Public Interfaces
  - New configuration (optional)
  - New ISslContextFactory interface
  - Important note about common SSL configurations
    - Final recommendation
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives
  - Modify existing SSLFactory to allow custom mechanism to load keystores only
  - Writing a new JSSE Security Provider

Status

Current state: Done

Discussion thread: here

Vote thread: here

JIRA: *CASSANDRA-16666*

Released: will be in 4.1, https://github.com/apache/cassandra/commit/24dcc280c2e442eea27e7129c4c948eb6199ed91

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

Motivation

Cassandra's adoption is growing day-by-day in the industry 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 Cassanddra. SSL/TLS communication is very critical part of those standards. While Cassandra supports SSL communication sometimes it becomes challenging to use out-of-the-box solution provided by Cassandra for SSL/TLS given the organization's needs.

Currently Cassandra creates the SSLContext via SSLFactory.java. SSLFactory is a final class with static methods and not overridable. The SSLFactory loads the keys and certs from the file based artifacts for the same. While this works for many, in the industry where security is stricter and contextual, this approach falls short. Many big organizations need flexibility to load the SSL artifacts from a custom resource (like custom Key Management Solution, HashiCorp Vault, Amazon KMS etc). While JSSE SecurityProvider architecture allows us flexibility to build our custom mechanisms to validate and process security artifacts, many times all we need is to build upon Java's existing extensibility that Trust/Key Manager interfaces provide to load keystores from various resources with the absence of any customized requirements on the Keys/Certificate formats.

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 Cassandra provides a way to customize SSLContext it would be a great lever and provide the ultimate extensibility for the Cassandra users.

My proposal here is to make the SSLContext creation pluggable/extensible and have the current SSLFactory.java implement an extensible interface. Of course we would need to also address requirements of Hot reloading of the SSLContext, currently implemented by this class for file based keystores.

Audience

Cassandra users.

Goals

- To allow users to customize the class for SSLContext creation via Cassandra Configuration and provide complete control of how SSLContext object is created
- Keep existing functionality like cache for the SSLContext objects, periodic thread to check for hot reloading etc "as-is"

Non-Goals

- To allow plugging in custom JSSE Provider. The requirement to plugin a custom JSSE Provider is out of scope for this CIP.
- To make SSLFactory#tlsInstanceProtocolSubstituion() pluggable because regardless of pluggability requirements it is just fetching the list of TLS protocol list
- To modify stress tool's configuration, LoaderOptions, some test classes to allow them having similar extensibility for the SSL Context generation in a custom way

Proposed Changes

I propose following changes,

1. Create a new Java interface 'ISslContextFactory'
2. Adding a new configuration like 'ssl_context_factory' with default value provided by newly created DefaultSslContextFactoryImpl.java
    a. Make necessary changes in Config.java and DatabaseDescriptor.java to load the new config if provided
3. Create a new DefaultSslContextFactoryImpl.java implementing the new ISslContextFactory interface
    a. This will be a 'final' class and will be considered internal without expectations of being extensible by any other public implementation
    b. This will be the only default implementation class for the SslContextFactory interface
4. Make changes in the existing SSLFactory and other classes to use the dependency on the newly created SslContextFactory
5. Make necessary changes to the existing code to allow triggering Hotreloading on the pluggable SslContextFactory
6. Make changes in the existing EncryptionOptions.java that uses 'keystore' to determine if it needs to enable encrypted connection

New or Changed Public Interfaces

New configuration (optional)

The new optional configuration key **under client/server_encryption_options** introduced to be configured **ONLY** in the case you need to use your own implementation for SSL Context creation.

New Configuration

```
 client/server_encryption_options:
        ssl_context_factory:
                class_name: org.apache.cassandra.security.YourSslContextFactoryImpl
                parameters:
                        key1: "value1"
                        key2: "value2"
                        key3: "value3"
```

New ISslContextFactory interface

ISslContextFactory

```
/**
 * The purpose of this interface is to provide pluggable mechanism for creating custom JSSE and Netty SSLContext
 * objects. Please use the Cassandra configuration key {@code ssl_context_factory} as part of {@code
 * client_encryption_options}/{@code server_encryption_options} and provide a custom class-name implementing
this
 * interface with parameters to be used to plugin a your own way to load the SSLContext.
 *
 * Implementation of this interface must have a constructor with argument of type {@code Map<String,Object>} to
allow
 * custom parameters, needed by the implementation, to be passed from the yaml configuration. Common SSL
 * configurations like {@code protocol, algorithm, cipher_suites, accepted_protocols, require_client_auth,
 * require_endpoint_verification, enabled, optional} will also be passed to that map by Cassanddra.
 *
 * Since on top of Netty, Cassandra is internally using JSSE SSLContext also for certain use-cases- this
interface
 * has methods for both.
 *
 * Below is an example of how to configure a custom implementation with parameters
 * <pre>
 * ssl_context_factory:
 *       class_name: org.apache.cassandra.security.YourSslContextFactoryImpl
 *       parameters:
```

```
 *         key1: "value1"
 *         key2: "value2"
 *         key3: "value3"
 * </pre>
 */
public interface ISslContextFactory
{
    /**
     * Creates JSSE SSLContext.
     *
     * @param buildTruststore {@code true} if the caller requires Truststore; {@code false} otherwise
     * @return
     * @throws SSLException in case the Ssl Context creation fails for some reason
     */
    SSLContext createJSSESslContext(boolean buildTruststore) throws SSLException;

    /**
     * Creates Netty's SslContext object.
     *
     * @param buildTruststore {@code true} if the caller requires Truststore; {@code false} otherwise
     * @param socketType {@link SocketType} for Netty's Inbound or Outbound channels
     * @param useOpenSsl {@code true} if openSsl is enabled;{@code false} otherwise
     * @param cipherFilter to allow Netty's cipher suite filtering, e.g.
     * {@link io.netty.handler.ssl.SslContextBuilder#ciphers(Iterable, CipherSuiteFilter)}
     * @return
     * @throws SSLException in case the Ssl Context creation fails for some reason
     */
    SslContext createNettySslContext(boolean buildTruststore, SocketType socketType,
                                     boolean useOpenSsl, CipherSuiteFilter cipherFilter) throws SSLException;

    /**
     * Initializes hot reloading of the security keys/certs. The implementation must guarantee this to be
thread safe.
     * @throws SSLException
     */
    void initHotReloading() throws SSLException;

    /**
     * Returns if any changes require the reloading of the SSL context returned by this factory.
     * This will be called by Cassandra's periodic polling for any potential changes that will reload the SSL
context
     * . However only newer connections established after the reload will use the reloaded SSL context.
     * @return
     */
    boolean shouldReload();

    /**
     * Returns if this factory uses private keystore.
     * @return {@code true} by default unless the implementation overrides this
     */
    default boolean hasKeystore() {
        return true;
    }

    /**
     * Returns the prepared list of accepted protocols.
     * @return array of protocol names suitable for passing to Netty's SslContextBuilder.protocols, or null if
the
     * default
     */
    List<String> getAcceptedProtocols();

    /**
     * Returns the list of cipher suites supported by the implementation.
     * @return
     */
    List<String> getCipherSuites();

    /**
     * Indicates if the process holds the inbound/listening end of the socket ({@link SSLFactory.
SocketType#SERVER})), or the
```

```
      * outbound side ({@link SSLFactory.SocketType#CLIENT}).
      */
     enum SocketType {
         SERVER, CLIENT;
     }
}
```

Important note about common SSL configurations

Currently the EncryptionOptions contains the keystore/truststore paths and other important SSL configuration parameters for the SSL connection (like ciphers, ssl protocol version, client-auth-required flag, endpoint verification flag etc). These "other important" configuration options we refer as the "common SSL configurations" in the title here.

Any implementation of the ISslContextFactory would still need to use these common configurations AND may require additional configuration parameters of their own. Since we are going to modify EncryptionOption to have one more option to define the pluggable implementation for the ISslContextFacgtory, this raises couple of questions which we need to address in terms of design-

1. Should not ideally we move out the common SSL configurations in the EncryptionOption and may be define a parent class like CommonEncryptionOption?
    a. Response: Ideally yes. While it is little weird to have conflicting configuration options in the EncryptionOption (i.e. file based keystore /truststore paths and pluggable sslcontext factory which mostly intended toward not using file based artifacts), practically we could still live with little imperfection. However we will be open to discuss if the community members feel it is best to have a parent class to move out 'common' ssl configurations between file based and pluggable sslcontext factory.
    b. Above stated reason results into the current design of the interface which requires EncryptionOptions to be passed to most of it's methods because the implementation needs to know the "common SSL configurations".
    c. **Changes would look like** this Git commit
2. Should we move to model of having Map<String,String> type as passing all the SSL configuration options to any implementation of the ISslContextFactory including the Default/Common one?
    a. Yes we could do that but for that we would have to "copy" all the common SSL configurations to the Map<String,String> while creating the instance of the ISslContextFactory's implementation including the Default one. This would also help make the imperfection of not adding CommonEncryptionOption suggested in the 1st question.
    b. Again, we would need community's input on the preference. **Changes would look like** this Git commit
    c. As you would notice from the above Git commit link, this option means we would need to move EncryptedOptions#acceptedProtocols() and EncryptionOptions#cipherSuiteArray() to ISslContextFactory interface

Final recommendation

Based on community input (discussion on the JIRA), having a Map makes more sense here. However a minor modification we have to do is it would be a Map<String,Object> instead of Map<String,String> since there are certain SSL configurations which are List type (e.g. cipher suites, accepted protocols etc).

Compatibility, Deprecation, and Migration Plan

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

No impact expected by existing users. Default implementation of SslContextFactory will ensure existing functionality is kept intact.

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

No need to phase out older behavior.

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

Not applicable

- *When will we remove the existing behavior?*

Not applicable

Test Plan

- Successful integration tests
- Successful local system test to use existing mechanism for the keystores and make sure the default SslContextFactory works for JSSE and Netty SSL Context creation
- For any additional requirements from the Cassandra community, will seek guidance on the discussion thread

Rejected Alternatives

Modify existing SSLFactory to allow custom mechanism to load keystores only

If we allow custom mechanism to load keystores from any source, it could still limit ability to customize other parameters in the SSLContext. This could become a limitation for a use-case. Delegating the creation of the whole SSLContext object is more flexible approach that can fit variety of use-cases.

Writing a new JSSE Security Provider

JSSE Security Provider is more suitable when we need to customize security implementation for validating certs (example SPIFFE) or other specific JSSE 'services' like digest etc. Here our primary goal is to make SSLContext creation only as a pluggable mechanism and the JSSE Security Provider seems a mis-fit solution for that.