KIP-12 - Kafka Sasl/Kerberos and SSL implementation

Status

Current state: *"Accepted"*

Discussion thread: *here*

JIRA: *here*

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

Motivation

The goal is to add sasl authentication capability to Kafka brokers and provide ssl for encryption.

Public Interfaces

- Channel wrapper for TransportLayer and AuthenticationLayer providing necessary handshake and authentication methods and also read (ByteBuffer buf) , write(ByteBuffer buf), write(ByteBuffer[] buf).
- TransportLayer is an interface for network transportLayer.
- PlainTextTransportLayer provides plain text socket channel methods
- SSLTransportLayer provides ssl handshake and read/write methods.
- Authenticator is an interface to providing client/server authentication.
- SaslServerAuthenticationLayer implements AuthenticationLayer, provides authentication methods for server side.
- SaslClientAuthenticationLayer implements AuthenticationLayer, provides client side authentication.

- User: This class will be used to get the remoteUserId and add it to the Session Object (https://issues.apache.org/jira/browse/KAFKA-1683)
- KafkaPrincipalToLocalPlugin: This is a pluggable class with a default implementation which translates a kerberos principal which looks like "testuser/node1.test.com@EXAMPLE.COM" to "testuser". Users can provide a their own customized version of PrincipalToLocalPlugin.
- AuthUtils: This class will consists of any utilities needed for SASL and other auth related methods.
- KerberosLoginFactory: It will use jaas config to login and generates a subject.
- Protocol accepts the protocol type (PLAINTEXT, SSL , PLAINTEXT+SASL, SSL+SASL)
    - PLAINTEXT (non-authenticated, non-encrypted)
        - This channel will provide exact behavior for communication channels as previous releases
    - SSL
        - SSL implementation. Authenticated principal in the session will be from the certificate presented or the peer host.
    - SASL+PLAINTEXT
        - SASL authentication will be used over plaintext channel. Once the sasl authentication established between client and server . Session will have client's principal as authenticated user. There won't be any wire encryption in this case as all the channel communication will be over plain text .
    - SASL+SSL

- SSL will be established initially and SASL authentication will be done over SSL. Once SASL authentication is established users principal will be used as authenticated user . This option is useful if users want to use SASL authentication (for example kerberos) with wire encryption.

- SecurityConfig , a config file for provider SecurityProtocol, SSL config and SASL mechanisms.
- BlockingChannel interface changes as it accepts the Protocol to create appropriate channels.

Proposed Changes

we will be using SASL to provide authentication and SSL to provider encryption in connection oriented protocols.

As part of SASL implementation we will be using JAAS config to read kerberos ticket and authenticate. More info on JAAS Config

http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html

Proposed JAAS Login config file will look like this.

Users needs to pass -Djava.security.auth.login.config=kafka_jaas.conf as part of JVM params .

This JAAS file along with Login.java can be used to login into LDAP or KERBEROS etc..

Here are some details on LdapLoginModule for JAAS https://docs.oracle.com/javase/8/docs/jre/api/security/jaas/spec/com/sun/security/auth/module/LdapLoginModule.html

Jaas Config

```
KafkaServer {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="/keytabs/kafka.keytab"
storeKey=true
useTicketCache=false
serviceName="kafka" // this will be used to connect to other brokers for replica management and also controller requests. This should be set to whatever principal that kafka brokers are running.
principal="kafka/_HOST@EXAMPLE.COM";
};
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="/keytabs/kafka.keytab"
storeKey=true
useTicketCache=false
serviceName="zookeeper"
principal="kafka@EXAMPLE.COM";
}


KafkaServer will be used to authenticate Kafka broker against kerberos
and Client section will be used for zkClient to access kerberos enabled zookeeper cluster.

KafkaClient {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="/keytabs/kafkaclient.keytab"
storeKey=true
useTicketCache=false
serviceName="kafka"
principal="kafkaproducer/_HOST@EXAMPLE.COM";
};

The above config is for any client ( producer, consumer) connecting to kerberos enabled Kafka cluster.
Here serviceName must match the principal name used under KafkaServer.
```

Channel

```
package org.apache.kafka.common.network;

public class Channel implements ScatteringByteChannel, GatheringByteChannel {
   private TransportLayer transportLayer;
   private Authenticator authenticator;

   public Channel(TransportLayer transportLayer, Authenticator authenticator) throws IOException

   /**
    * returns user principal for the session
    * Incase of PLAINTEXT and No Authentication returns ANONYMOUS as the userPrincipal
    * If SSL used without any SASL Authentication returns SSLSession.peerPrincipal
    */
    public UserPrincipal userPrincipal() {
       return authenticator.userPrincipal();
    }

    /**
     * starts transportLayer handshake
     * If transportLayer handshake finishes, than initiates AuthenticationLayer.authenticate
     * @returns 0 if handshake and authentication finishes otherwise returns appropriate SelctionKey.OP
     */
    public int connect(boolean read, boolean write) throws IOException

    /**
     *
     */
    public void blockingConnect(long timeout) throws IOException

    @Override
    public int write(ByteBuffer src) throws IOException

    @Override
    public int read(ByteBuffer dst) throws IOException

    /* returns the socketChannel */
    public SocketChannel socketChannel();

        @Override
    public long write(ByteBuffer[] srcs) throws IOException

    @Override
    public long write(ByteBuffer[] srcs, int offset, int length) throws IOException

    @Override
    public int read(ByteBuffer dst) throws IOException

    @Override
    public long read(ByteBuffer[] dsts) throws IOException

    @Override
    public long read(ByteBuffer[] dsts, int offset, int length) throws IOException

    public boolean finishConnect() throws IOException

    public DataInputStream getInputStream() throws IOException

    public DataOutputStream getOutputStream() throws IOException

    public void close() throws IOException

 }
```

TransportLayer

```
public interface TransportLayer {

    /**
     * Closes this channel
     *
     * @throws IOException If and I/O error occurs
     */
    void close() throws IOException;


    /**
     * Tells wheather or not this channel is open.
     */
    boolean isOpen();

    /**
     * Writes a sequence of bytes to this channel from the given buffer.
     */
    int write(ByteBuffer src) throws IOException;

    long write(ByteBuffer[] srcs) throws IOException;

    long write(ByteBuffer[] srcs, int offset, int length) throws IOException;

    int read(ByteBuffer dst) throws IOException;

    long read(ByteBuffer[] dsts) throws IOException;

    long read(ByteBuffer[] dsts, int offset, int length) throws IOException;

    boolean isReady();

    boolean finishConnect() throws IOException;

    SocketChannel socketChannel();

    /**
     * Performs SSL handshake hence is a no-op for the non-secure
     * implementation
     * @param read Unused in non-secure implementation
     * @param write Unused in non-secure implementation
     * @return Always return 0
     * @throws IOException
    */
    int handshake(boolean read, boolean write) throws IOException;

    DataInputStream inStream() throws IOException;

    DataOutputStream outStream() throws IOException;

    boolean flush(ByteBuffer buffer) throws IOException;

    Principal getPeerPrincipal();
}
```

PlainTextTransportLayer

```
public class PlainTextTransportLayer implements TransportLayer {
   public PlainTextTransportLayer(SocketChannel socketChannel) throws IOException
}
```

SSLTransportLayer

```
public class SSLTransportLayer implements TransportLayer {
        public SSLTransportLayer(SocketChannel socketChannel, SSLEngine sslEngine) throws IOException
}
```

Authenticator

```
public interface Authenticator {

    /**
     * Closes any resources
     *
     * @throws IOException if any I/O error occurs
     */
    void close() throws IOException;

    /**
     *
     * @throws IOException
     */
    void init() throws IOException;

    /**
     * Returns UserPrincipal after authentication is established
     */
    UserPrincipal userPrincipal();


    /**
     * Does authentication in non-blocking way and returns SelectionKey.OP if further communication needed
     */
    int authenticate(boolean read, boolean write) throws IOException;

    /**
     * returns true if authentication is complete otherwise returns false;
     */

    boolean isComplete();

}
```

DefaultAuthenticator

```
public class DefaultAuthenticator implements Authenticator {

    TransportLayer transportLayer;

    public DefaultAuthenticator(TransportLayer transportLayer) {
        this.transportLayer = transportLayer;
    }

    public void init() {}

    public int authenticate(boolean read, boolean write) throws IOException {
        return 0;
    }

    /** returns peer host incase of SSL */
    public UserPrincipal userPrincipal() {
        return new UserPrincipal(transportLayer.getPeerPrincipal().toString());
    }

    public void close() throws IOException {}

    public boolean isComplete() {
        return true;
    }
}
```

SaslServerAuthenticator

```
public class SaslServerAuthenticator implements Authenticator {

    public SaslServerAuthenticator(final Subject subject, TransportLayer transportLayer) {
    }
}
```

SaslClientAuthenticator

```
public class SaslServerAuthenticator implements Authenticator {
   public SaslServerAuthenticator(final Subject subject, TransportLayer transportLayer) {
   }
}
```

Callbacks for Login , SaslServer, SaslClient

Callbacks for the above modules will help in grabbing the users authentication information.

AuthUtils

```
public class AuthUtils {
   /**
     * Construct a JAAS configuration object per kafka jaas configuration file
     * @param storm_conf Storm configuration
     * @return JAAS configuration object
     */
    public static Configuration getConfiguration(String jaasConfigFilePath)

    public static String getJaasConfig(String loginContextName, String key) throws IOException

    public static String getDefaultRealm()

}
```

Login

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

public class Login {
private volatile Subject subject = null;

//logs in based on jaas conf and starts a thread to renew it .
public Login(final String loginContextName)
            throws LoginException

// returns generated subject after the login.
public Subject getSubject()


}
```

KerberosLoginManager

```
package kafka.common

object KerberosLoginManager {
   var kerberosLogin: Login

   //does kerberos login and generates the subject
   def init()

   // returns kerberosLogin.getSubject()
   def subject()

   def shutdown()

}
```

SASL Authentication exchange

KafkaClient

1) KafkaClient picks the principal it wants to use by looking at KafkaClient jaas config (example above).

2) Authenticates against KDC (kerberos) using the KafkaClient principal.

3) KafkaClient constructs the service principal name based on the jaas config serviceName

4) KafkaClient initiates challenge/response with KafkaBroker along with KafkaClient principal and service principal . Depending on the KafkaBroker response these challenge/response might continue until it receives COMPLETE from the KafkaBroker.

KafkaBroker

1) KafkaBroker will accept the connection and accepts the client and service principal

2) checks if the service principal is same as the one KafkaBroker running with.

3) KafkaBroker accepts/rejects the clients token

4) Returns the response to the client if its authenticated or not.

5) Once client is authenticated they can send Kafka requests.

Security Config

SecurityConfig will be shared across clients and brokers. If not provided communication channels fall back to PLAINTEXT . Here are proposed configs

```
sasl.authentication.mechanism (KERBEROS will be supported for revision1)
ssl.protocol
ssl.cipher.suites
ssl.enabled.protocols
ssl.keystore.type
ssl.keystore.location
ssl.keystore.password
ssl.key.password
ssl.truststore.type
ssl.truststore.location
ssl.truststore.password
ssl.client.require.cert
ssl.keymanager.algorithm
ssl.trustmanager.algorithm
```

Compatibility, Deprecation, and Migration Plan

As per previous security discussions and multiport work being done as part of this JIRA

Unable to render Jira issues macro, execution error.

,

Users/Clients can still communicate with non-secure/non-sasl kafka brokers.

Open Questions

1) Users should be configuring -Djava.security.auth.login.config=jass_file.conf . after setting authentication.enable to true in server and also in clients. Any objections on this approach?

2) We need to pass SecurityConfig to ReplicaManager -> ReplicaFetcherManager -> ReplicaFetcherThread -> SimpleConsumer -> BlockingChannel. This is a necessary as BlockingChannel can based on the

Protocol.securityProtocol initiate the appropriate Channel described earlier. Any better approach passing down this information to BlockingChannel to create appropriate TransportLayer and AuthenticationLayer.

3) Similarly Producer and Consumer does KerberosLoginManager.init() with "KafkaClient" section if authentication.enable set to true and also passes Protocol to BlockingChannel. Any alternatives or objections on this approach?

Rejected Alternatives

1) Using GSS-API as an authentication layer and security layer. This can only be used with Kerberos.

2) Using SASL to provide encryption of network data. After talking to other projects like HDFS (HBASE as well) they noticed 10x slow down when encryption is enabled on SASL. Hence the reason to separate transportLayer and authentication layer . SASL as authentication users can choose PLAINTEXT for no encryption and SSL for encryption and still be performant.

3) Using TLS for kerberos authentication. https://tools.ietf.org/html/rfc6251 . In this case clients expects server to be running with "host/hostname@realm" keytab. Here "host" cannot be changed.

HDFS noticed security issues as well with TLS for kerberos.