KIP-255 OAuth Authentication via SASL/OAUTHBEARER

Status

Current state: *Adopted (in 2.0)*

Discussion thread: here

JIRA: KAFKA-6562

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

Motivation

The ability to authenticate to Kafka with an OAuth 2 Access Token is desirable given the popularity of OAuth. OAuth 2 is a flexible framework with multiple ways of doing the same thing (for example, getting a public key to confirm a digital signature), so pluggable implementations – and flexibility in their configuration – is a requirement; the introduction of KIP-86: Configurable SASL callback handlers provides the necessary flexibility. "OAUTHBEARER" is the SASL mechanism for OAuth 2.

This KIP proposes to add the following functionality related to SASL/OAUTHBEARER:

- Allow clients (both brokers when SASL/OAUTHBEARER is the inter-broker protocol as well as non-broker clients) to flexibly retrieve an access token from an OAuth 2 authorization server based on the declaration of a custom login `CallbackHandler` implementation and have that access token transparently and automatically transmitted to a broker for authentication.
- Allow brokers to flexibly validate provided access tokens when a client establishes a connection based on the declaration of a custom SASL Server `CallbackHandler` implementation.
- Provide implementations of the above retrieval and validation features based on an unsecured JSON Web Token that function out-of-the-box with minimal configuration required (i.e. implementations of the two types of callback handlers mentioned above will be used by default with no need to explicitly declare them). This unsecured functionality serves two purposes: first, it provides a way for SASL/OAUTHBEARER to be used in development scenarios out-of-the-box with no OAuth 2 infrastructure required; and second, it provides a way to test the SASL implementation itself. See Rejected Alternatives: Motivation.
- Allow clients (both brokers when SASL/OAUTHBEARER is the inter-broker protocol as well as non-broker clients) to transparently retrieve a new access token in the background before the existing access token expires in case the client has to open new connections. Any existing connections will remain unaffected by this "token refresh" functionality as long as the connection remains intact, but new connections from the same client will always use the latest access token (either the initial one or the one that was most recently retrieved by the token refresh functionality, if any). This is how Kerberos-authenticated connections work with respect to ticket expiration. This KIP does not attempt to unify the refresh-related code for the OAUTHBEARER and GSSAPI mechanisms, but it does include code that suggests a potential path forward if this unification is desired in the future.

Note that the access token can be made available to the broker for authorization decisions due to KIP-189: Improve principal builder interface and add support for SASL (by exposing the access token via a negotiated property on the `SaslServer` implementation), but detailed discussion of this possibility is outside the scope of this proposal. It is noted, however, that if access tokens are somehow used for authorization decisions, it is conceivable due to the long-lived nature of Kafka connections that authorization decisions will sometimes be made using expired access tokens. For example, it is up to the broker to validate the token upon authentication, but the token will not be replaced for that particular connection as long as it remains intact; if the token expires in an hour then authorization decisions for that first hour will be made using the still-valid token, but after an hour the expired token would remain associated with the connection, and authorization decisions from that point forward for that particular connection would be made using the expired token. This would have to be addressed via a separate KIP if it turns out to be problematic, but that seems unlikely (code signing certificates that have been timestamped remain valid after their expiration, for example, and access tokens are indeed timestamped). Another related issue that would need to be addressed is how to revoke authorizations. Connections are long-lived, and bearer tokens are immutable, so a mechanism to evolve or revoke permissions over time would have to exist. Again, this is outside the scope of this KIP.

Finally, note that the implementation of flexible, substitution-aware configuration that was originally proposed in an early draft of this KIP was at first deemed more generally useful and was separated out into its own KIP-269: Substitution Within Configuration Values, but that KIP is likely to be rejected /moved to the inactive list and is not required for this KIP (see Rejected Alternatives: Substitution Within Configuration Values).

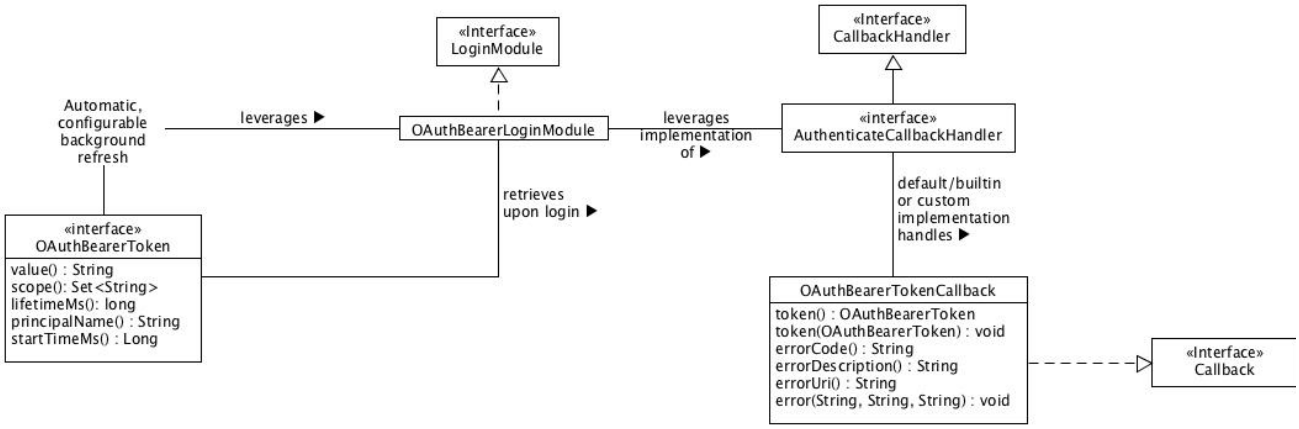
Public Interfaces

The public interface for this KIP consists of 3 Java classes and 1 Java interface along with various configuration possibilities. The following sections define these public-facing parts of this KIP, including an overall UML diagram and important code details (with Javadoc) where appropriate.

OAuth Bearer Tokens and Token Retrieval

See Rejected Alternatives: Explicit Configuration of Token Refresh Class

See Rejected Alternatives: Callback Handlers and Callbacks



We define `org.apache.kafka.common.security.oauthbearer.OAuthBearerToken` to be the interface that all OAuth 2 bearer tokens must implement within the context of Kafka's SASL/OAUTHBEARER implementation. Scenarios that leverage open source JWT/JWS/JWE implementations must wrap the library's implementation of a token to implement this interface.

The `org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule` class is the JAAS login module that is declared in the JAAS configuration. When a client (whether a non-broker client or a broker when SASL/OAUTHBEARER is the inter-broker protocol) connects to Kafka the `OAuthBearerLoginModule` instance asks its configured `AuthenticateCallbackHandler` implementation to handle an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerTokenCallback` and retrieve/return an instance of `OAuthBearerToken`. A default, builtin `AuthenticateCallbackHandler` implementation creates an unsecured token as defined by the JAAS module options – see below for configuration details – when another implementation is not explicitly specified. Production use cases will require writing an implementation of `AuthenticateCallbackHandler` that can handle an instance of `OAuthBearerTokenCallback` and declaring it via either the `sasl.login.callback.handler.class` configuration option for a non-broker client or via the `listener.name.sasl_ssl.oauthbearer.sasl.login.callback.handler.class` configuration option for brokers (when SASL/OAUTHBEARER is the inter-broker protocol).

Here are the parameters that can be provided as part of the JAAS configuration to determine the claims that appear in an unsecured OAuth Bearer Token generated by the default, out-of-the-box `AuthenticateCallbackHandler` implementation.

JAAS Module Option for Unsecured Token Retrieval	Documentation
`unsecuredLoginStringClaim_<claimname>="value"`	Creates a String claim with the given name and value. Any valid claim name can be specified except '`iat`' and '`exp`' (these are automatically generated).
`unsecuredLoginNumberClaim_<claimname>="value"`	Creates a Number claim with the given name and value. Any valid claim name can be specified except '`iat`' and '`exp`' (these are automatically generated).
`unsecuredLoginListClaim_<claimname>="value"`	Creates a String List claim with the given name and values parsed from the given value where the first character is taken as the delimiter. For example: `unsecuredLoginListClaim_fubar="|value1|value2"`. Any valid claim name can be specified except '`iat`' and '`exp`' (these are automatically generated).
`unsecuredLoginPrincipalClaimName`	Set to a custom claim name if you wish the name of the String claim holding the principal name to be something other than '`sub`'.
`unsecuredLoginLifetimeSeconds`	Set to an integer value if the token expiration is to be set to something other than the default value of 3600 seconds (which is 1 hour). The '`exp`' claim will be set to reflect the expiration time.
`unsecuredLoginScopeClaimName`	Set to a custom claim name if you wish the name of the String or String List claim holding any token scope to be something other than '`scope`'.

Here is a typical, basic JAAS configuration for a client leveraging unsecured SASL/OAUTHBEARER authentication:

```
KafkaClient {
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule Required
    unsecuredLoginStringClaim_sub="thePrincipalName";
};
```

An implementation of the `org.apache.kafka.common.security.auth.Login` interface specific to the OAUTHBEARER mechanism is automatically applied; it periodically refreshes any token before it expires so that the client can continue to make connections to brokers. The parameters that impact how the refresh algorithm operates are specified as part of the producer/consumer/broker configuration; they are as follows (the defaults are generally reasonable, so explicit configuration may not be necessary):

Producer/Consumer/Broker Configuration Property	Documentation
`sasl.login.refresh.window.factor`	Login refresh thread will sleep until the specified window factor relative to the credential's lifetime has been reached, at which time it will try to refresh the credential. Legal values are between 0.5 (50%) and 1.0 (100%) inclusive; a default value of 0.8 (80%) is used if no value is specified. Currently applies only to OAUTHBEARER.
`sasl.login.refresh.window.jitter`	The maximum amount of random jitter relative to the credential's lifetime that is added to the login refresh thread's sleep time. Legal values are between 0 and 0.25 (25%) inclusive; a default value of 0.05 (5%) is used if no value is specified. Currently applies only to OAUTHBEARER.
`sasl.login.refresh.min.period.seconds`	The desired minimum time for the login refresh thread to wait before refreshing a credential, in seconds. Legal values are between 0 and 900 (15 minutes); a default value of 60 (1 minute) is used if no value is specified. This value and `sasl.login.refresh.buffer.seconds` are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.
`sasl.login.refresh.buffer.seconds`	The amount of buffer time before credential expiration to maintain when refreshing a credential, in seconds. If a refresh would otherwise occur closer to expiration than the number of buffer seconds then the refresh will be moved up to maintain as much of the buffer time as possible, overriding all other considerations. Legal values are between 0 and 3600 (1 hour); a default value of 300 (5 minutes) is used if no value is specified. This value and `sasl.login.refresh.min.period.seconds` are both ignored if their sum exceeds the remaining lifetime of a credential. Currently applies only to OAUTHBEARER.

org.apache.kafka.common.security.oauthbearer.OAuthBearerToken

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

/**
 * The <code>b64token</code> value as defined in
 * <a href="https://tools.ietf.org/html/rfc6750#section-2.1">RFC 6750 Section
 * 2.1</a> along with the token's specific scope and lifetime and principal
 * name.
 * <p>
 * A network request would be required to re-hydrate an opaque token, and that
 * could result in (for example) an {@code IOException}, but retrievers for
 * various attributes ({@link #scope()}, {@link #lifetimeMs()}, etc.) declare no
 * exceptions. Therefore, if a network request is required for any of these
 * retriever methods, that request could be performed at construction time so
 * that the various attributes can be reliably provided thereafter. For example,
 * a constructor might declare {@code throws IOException} in such a case.
 * Alternatively, the retrievers could throw unchecked exceptions.
 *
 * @see <a href="https://tools.ietf.org/html/rfc6749#section-1.4">RFC 6749
 *      Section 1.4</a> and
 *      <a href="https://tools.ietf.org/html/rfc6750#section-2.1">RFC 6750
 *      Section 2.1</a>
 */
public interface OAuthBearerToken {
    /**
     * The <code>b64token</code> value as defined in
     * <a href="https://tools.ietf.org/html/rfc6750#section-2.1">RFC 6750 Section
```

```
     * 2.1</a>
     *
     * @return <code>b64token</code> value as defined in
     *         <a href="https://tools.ietf.org/html/rfc6750#section-2.1">RFC 6750
     *         Section 2.1</a>
     */
    String value();

    /**
     * The token's scope of access, as per
     * <a href="https://tools.ietf.org/html/rfc6749#section-1.4">RFC 6749 Section
     * 1.4</a>
     *
     * @return the token's (always non-null but potentially empty) scope of access,
     *         as per <a href="https://tools.ietf.org/html/rfc6749#section-1.4">RFC
     *         6749 Section 1.4</a>. Note that all values in the returned set will
     *         be trimmed of preceding and trailing whitespace, and the result will
     *         never contain the empty string.
     */
    Set<String> scope();

    /**
     * The token's lifetime, expressed as the number of milliseconds since the
     * epoch, as per <a href="https://tools.ietf.org/html/rfc6749#section-1.4">RFC
     * 6749 Section 1.4</a>
     *
     * @return the token'slifetime, expressed as the number of milliseconds since
     *         the epoch, as per
     *         <a href="https://tools.ietf.org/html/rfc6749#section-1.4">RFC 6749
     *         Section 1.4</a>.
     */
    long lifetimeMs();

    /**
     * The name of the principal to which this credential applies
     *
     * @return the always non-null/non-empty principal name
     */
    String principalName();

    /**
     * When the credential became valid, in terms of the number of milliseconds
     * since the epoch, if known, otherwise null. An expiring credential may not
     * necessarily indicate when it was created -- just when it expires -- so we
     * need to support a null return value here.
     *
     * @return the time when the credential became valid, in terms of the number of
     *         milliseconds since the epoch, if known, otherwise null
     */
    Long startTimeMs();
}
```

org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule

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

/**
 * The {@code LoginModule} for the SASL/OAUTHBEARER mechanism. When a client
 * (whether a non-broker client or a broker when SASL/OAUTHBEARER is the
 * inter-broker protocol) connects to Kafka the {@code OAuthBearerLoginModule}
 * instance asks its configured {@link AuthenticateCallbackHandler}
 * implementation to handle an instance of {@link OAuthBearerTokenCallback} and
 * return an instance of {@link OAuthBearerToken}. A default, builtin
 * {@link AuthenticateCallbackHandler} implementation creates an unsecured token
 * as defined by these JAAS module options:
 * <table>
 * <tr>
 * <th>JAAS Module Option for Unsecured Token Retrieval</th>
 * <th>Documentation</th>
```

```
* </tr>
* <tr>
* <td>{@code unsecuredLoginStringClaim_<claimname>="value"}</td>
* <td>Creates a {@code String} claim with the given name and value. Any valid
* claim name can be specified except '{@code iat}' and '{@code exp}' (these are
* automatically generated).</td>
* </tr>
* <tr>
* <td>{@code unsecuredLoginNumberClaim_<claimname>="value"}</td>
* <td>Creates a {@code Number} claim with the given name and value. Any valid
* claim name can be specified except '{@code iat}' and '{@code exp}' (these are
* automatically generated).</td>
* </tr>
* <tr>
* <td>{@code unsecuredLoginListClaim_<claimname>="value"}</td>
* <td>Creates a {@code String List} claim with the given name and values parsed
* from the given value where the first character is taken as the delimiter. For
* example: {@code unsecuredLoginListClaim_fubar="|value1|value2"}. Any valid
* claim name can be specified except '{@code iat}' and '{@code exp}' (these are
* automatically generated).</td>
* </tr>
* <tr>
* <td>{@code unsecuredLoginPrincipalClaimName}</td>
* <td>Set to a custom claim name if you wish the name of the String claim
* holding the principal name to be something other than '{@code sub}'.</td>
* </tr>
* <tr>
* <td>{@code unsecuredLoginLifetimeSeconds}</td>
* <td>Set to an integer value if the token expiration is to be set to something
* other than the default value of 3600 seconds (which is 1 hour). The
* '{@code exp}' claim will be set to reflect the expiration time.</td>
* </tr>
* <tr>
* <td>{@code unsecuredLoginScopeClaimName}</td>
* <td>Set to a custom claim name if you wish the name of the String or String
* List claim holding any token scope to be something other than
* '{@code scope}'.</td>
* </tr>
* </table>
* Production use cases will require writing an implementation of
* {@link AuthenticateCallbackHandler} that can handle an instance of
* {@link OAuthBearerTokenCallback} and declaring it via either the
* {@code sasl.login.callback.handler.class} configuration option for a
* non-broker client or via the
* {@code listener.name.sasl_ssl.oauthbearer.sasl.login.callback.handler.class}
* configuration option for brokers (when SASL/OAUTHBEARER is the inter-broker
* protocol).
* <p>
* Here is a typical, basic JAAS configuration for a client leveraging unsecured
* SASL/OAUTHBEARER authentication:
*
* <pre>
* KafkaClient {
*      org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule Required
*      unsecuredLoginStringClaim_sub="thePrincipalName";
* };
* </pre>
*
* An implementation of the {@link Login} interface specific to the
* {@code OAUTHBEARER} mechanism is automatically applied; it periodically
* refreshes any token before it expires so that the client can continue to make
* connections to brokers. The parameters that impact how the refresh algorithm
* operates are specified as part of the producer/consumer/broker configuration
* are as follows. See the documentation for these properties elsewhere for
* details.
* <table>
* <tr>
* <th>Producer/Consumer/Broker Configuration Property</th>
* </tr>
* <tr>
* <td>{@code sasl.login.refresh.window.factor}</td>
```

```
 * </tr>
 * <tr>
 * <td>{@code sasl.login.refresh.window.jitter}</td>
 * </tr>
 * <tr>
 * <td>{@code sasl.login.refresh.min.period.seconds}</td>
 * </tr>
 * <tr>
 * <td>{@code sasl.login.refresh.min.buffer.seconds}</td>
 * </tr>
 * </table>
 * When a broker accepts a SASL/OAUTHBEARER connection the instance of the
 * builtin {@code SaslServer} implementation asks its configured
 * {@link AuthenticateCallbackHandler} implementation to handle an instance of
 * {@link OAuthBearerValidatorCallback} constructed with the OAuth 2 Bearer
 * Token's compact serialization and return an instance of
 * {@link OAuthBearerToken} if the value validates. A default, builtin
 * {@link AuthenticateCallbackHandler} implementation validates an unsecured
 * token as defined by these JAAS module options:
 * <table>
 * <tr>
 * <th>JAAS Module Option for Unsecured Token Validation</th>
 * <th>Documentation</th>
 * </tr>
 * <tr>
 * <td>{@code unsecuredValidatorPrincipalClaimName="value"}</td>
 * <td>Set to a non-empty value if you wish a particular String claim holding a
 * principal name to be checked for existence; the default is to check for the
 * existence of the '{@code sub}' claim.</td>
 * </tr>
 * <tr>
 * <td>{@code unsecuredValidatorScopeClaimName="value"}</td>
 * <td>Set to a custom claim name if you wish the name of the String or String
 * List claim holding any token scope to be something other than
 * '{@code scope}'.</td>
 * </tr>
 * <tr>
 * <td>{@code unsecuredValidatorRequiredScope="value"}</td>
 * <td>Set to a space-delimited list of scope values if you wish the
 * String/String List claim holding the token scope to be checked to make sure
 * it contains certain values.</td>
 * </tr>
 * <tr>
 * <td>{@code unsecuredValidatorAllowableClockSkewMs="value"}</td>
 * <td>Set to a positive integer value if you wish to allow up to some number of
 * positive milliseconds of clock skew (the default is 0).</td>
 * </tr>
 * </table>
 * Here is a typical, basic JAAS configuration for a broker leveraging unsecured
 * SASL/OAUTHBEARER validation:
 *
 * <pre>
 * KafkaServer {
 *      org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule Required
 *      unsecuredLoginStringClaim_sub="thePrincipalName";
 * };
 * </pre>
 *
 * Production use cases will require writing an implementation of
 * {@link AuthenticateCallbackHandler} that can handle an instance of
 * {@link OAuthBearerValidatorCallback} and declaring it via the
 * {@code listener.name.sasl_ssl.oauthbearer.sasl.server.callback.handler.class}
 * configuration option.
 *
 * @see SaslConfigs#SASL_LOGIN_REFRESH_WINDOW_FACTOR_DOC
 * @see SaslConfigs#SASL_LOGIN_REFRESH_WINDOW_JITTER_DOC
 * @see SaslConfigs#SASL_LOGIN_REFRESH_MIN_PERIOD_SECONDS_DOC
 * @see SaslConfigs#SASL_LOGIN_REFRESH_MIN_BUFFER_SECONDS_DOC
 */
public class OAuthBearerLoginModule implements LoginModule {
    static {
```

```
        OAuthBearerSaslClientProvider.initialize(); // not part of public API
        OAuthBearerSaslServerProvider.initialize(); // not part of public API
    }

    // etc...
}
```

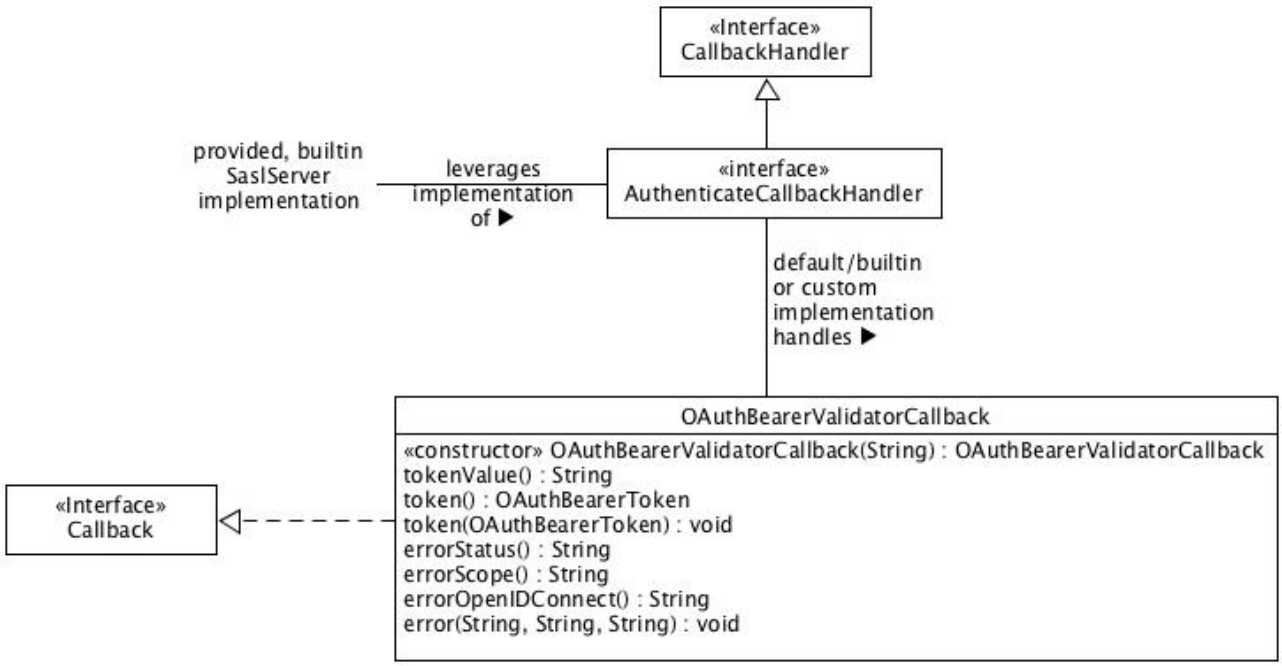
org.apache.kafka.common.security.oauthbearer.OAuthBearerTokenCallback

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

/**
 * A {@code Callback} for use by the {@code SaslClient} and {@code Login}
 * implementations when they require an OAuth 2 bearer token. Callback handlers
 * should use the {@link #error(String, String, String)} method to communicate
 * errors returned by the authorization server as per
 * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
 * 2.0 Authorization Framework</a>. Callback handlers should communicate other
 * problems by raising an {@code IOException}.
 */
public class OAuthBearerTokenCallback implements Callback {
    private OAuthBearerToken token = null;
    private String errorCode = null;
    private String errorDescription = null;
    private String errorUri = null;

    /**
     * Return the (potentially null) token
     *
     * @return the (potentially null) token
     */
    public OAuthBearerToken token() {
        return token;
    }

    /**
     * Return the (always non-empty) error code as per
     * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
     * 2.0 Authorization Framework</a>.
     *
     * @return the (always non-empty) error code
     */
    public String errorCode() {
        return errorCode;
    }

    /**
     * Return the (potentially null) error description as per
     * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
     * 2.0 Authorization Framework</a>.
     *
     * @return the (potentially null) error description
     */
    public String errorDescription() {
        return errorDescription;
    }

    /**
     * Return the (potentially null) error URI as per
     * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
     * 2.0 Authorization Framework</a>.
     *
     * @return the (potentially null) error URI
     */
    public String errorUri() {
        return errorUri;
    }

    /**
```

```
     * Set the token. All error-related values are cleared.
     *
     * @param token
     *            the mandatory token to set
     */
    public void token(OAuthBearerToken token) {
        this.token = Objects.requireNonNull(token);
        this.errorCode = null;
        this.errorDescription = null;
        this.errorUri = null;
    }

    /**
     * Set the error values as per
     * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
     * 2.0 Authorization Framework</a>. Any token is cleared.
     *
     * @param errorCode
     *            the mandatory error code to set
     * @param errorDescription
     *            the optional error description to set
     * @param errorCode
     *            the optional error URI to set
     */
    public void error(String errorCode, String errorDescription, String errorUri) {
        if (Objects.requireNonNull(errorCode).isEmpty())
            throw new IllegalArgumentException("error code must not be empty");
        this.errorCode = errorCode;
        this.errorDescription = errorDescription;
        this.errorUri = errorUri;
        this.token = null;
    }
}
```

Token Validation

See Rejected Alternatives: Callback Handlers and Callbacks



We define the `org.apache.kafka.common.security.oauthbearer.OAuthBearerValidatorCallback` class as the callback class for communicating that we want to validate a bearer token compact serialization provided by the connecting client. When a broker accepts a SASL /OAUTHBEARER connection the instance of the builtin `SaslServer` implementation asks its configured `AuthenticateCallbackHandler` implementation to handle an instance of `OAuthBearerValidatorCallback` constructed with the OAuth 2 Bearer Token's compact serialization and return an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerToken` if the compact serialization validates. A default, builtin `AuthenticateCallbackHandler` implementation validates an unsecured token as defined by these JAAS module options:

JAAS Module Option for Unsecured Token Validation	Documentation
`unsecuredValidatorPrincipalClaimName="value"`	Set to a non-empty value if you wish a particular String claim holding a principal name to be checked for existence; the default is to check for the existence of the '`sub`' claim.
`unsecuredValidatorScopeClaimName="value"`	Set to a custom claim name if you wish the name of the String or String List claim holding any token scope to be something other than '`scope`'
`unsecuredValidatorRequiredScope="value"`	Set to a space-delimited list of scope values if you wish the String/String List claim holding the token scope to be checked to make sure it contains certain values.
`unsecuredValidatorAllowableClockSkewMs="value"`	Set to a positive integer value if you wish to allow up to some number of positive milliseconds of clock skew (the default is 0).

Here is a typical, basic JAAS configuration for a broker leveraging unsecured SASL/OAUTHBEARER authentication:

```
KafkaServer {
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule Required
    unsecuredLoginStringClaim_sub="thePrincipalName";
};
```

Production use cases will require writing an implementation of `AuthenticateCallbackHandler` that can handle an instance of `OAuthBearerValidatorCallback` and declaring it via the `listener.name.sasl_ssl.oauthbearer.sasl.server.callback.handler.class` configuration option.

The validated token will be available as a negotiated property on the `SaslServer` instance with the key **`OAUTHBEARER.token`** so it can be used for authorization as per KIP-189: Improve principal builder interface and add support for SASL. Note that the implementation of `SaslServer` is not part of the public interface – just the key where it makes the validated token available.

org.apache.kafka.common.security.oauthbearer.OAuthBearerValidatorCallback

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

/**
 * A {@code Callback} for use by the {@code SaslServer} implementation when it
 * needs to provide an OAuth 2 bearer token compact serialization for
 * validation. Callback handlers should use the
 * {@link #error(String, String, String)} method to communicate errors back to
 * the SASL Client as per
 * <a href="https://tools.ietf.org/html/rfc6749#section-5.2">RFC 6749: The OAuth
 * 2.0 Authorization Framework</a> and the <a href=
 * "https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#extensions-error">IANA
 * OAuth Extensions Error Registry</a>. Callback handlers should communicate
 * other problems by raising an {@code IOException}.
 */
public class OAuthBearerValidatorCallback implements Callback {
    private final String tokenValue;
    private OAuthBearerToken token = null;
    private String errorStatus = null;
    private String errorScope = null;
    private String errorOpenIDConfiguration = null;

    /**
     * Constructor
     *
     * @param tokenValue
     *            the mandatory/non-blank token value
     */
    public OAuthBearerValidatorCallback(String tokenValue) {
        if (Objects.requireNonNull(tokenValue).isEmpty())
            throw new IllegalArgumentException("token value must not be empty");
        this.tokenValue = tokenValue;
```

```
}

/**
 * Return the (always non-null) token value
 *
 * @return the (always non-null) token value
 */
public String tokenValue() {
    return tokenValue;
}

/**
 * Return the (potentially null) token
 *
 * @return the (potentially null) token
 */
public OAuthBearerToken token() {
    return token;
}

/**
 * Return the (potentially null) error status value as per
 * <a href="https://tools.ietf.org/html/rfc7628#section-3.2.2">RFC 7628: A Set
 * of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth</a>
 * and the <a href=
 * "https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#extensions-error">IANA
 * OAuth Extensions Error Registry</a>.
 *
 * @return the (potentially null) error status value
 */
public String errorStatus() {
    return errorStatus;
}

/**
 * Return the (potentially null) error scope value as per
 * <a href="https://tools.ietf.org/html/rfc7628#section-3.2.2">RFC 7628: A Set
 * of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth</a>.
 *
 * @return the (potentially null) error scope value
 */
public String errorScope() {
    return errorScope;
}

/**
 * Return the (potentially null) error openid-configuration value as per
 * <a href="https://tools.ietf.org/html/rfc7628#section-3.2.2">RFC 7628: A Set
 * of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth</a>.
 *
 * @return the (potentially null) error openid-configuration value
 */
public String errorOpenIDConfiguration() {
    return errorOpenIDConfiguration;
}

/**
 * Set the token. The token value is unchanged and is expected to match the
 * provided token's value. All error values are cleared.
 *
 * @param token
 *            the mandatory token to set
 */
public void token(OAuthBearerToken token) {
    this.token = Objects.requireNonNull(token);
    this.errorStatus = null;
    this.errorScope = null;
    this.errorOpenIDConfiguration = null;
}

/**
```

```
     * Set the error values as per
     * <a href="https://tools.ietf.org/html/rfc7628#section-3.2.2">RFC 7628: A Set
     * of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth</a>.
     * Any token is cleared.
     *
     * @param errorStatus
     *            the mandatory error status value from the <a href=
     *            "https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#extensions-error"
>IANA
     *            OAuth Extensions Error Registry</a> to set
     * @param errorScope
     *            the optional error scope value to set
     * @param errorStatus
     *            the optional error openid-configuration value to set
     */
    public void error(String errorStatus, String errorScope, String errorOpenIDConfiguration) {
        if (Objects.requireNonNull(errorStatus).isEmpty())
            throw new IllegalArgumentException("error status must not be empty");
        this.errorStatus = errorStatus;
        this.errorScope = errorScope;
        this.errorOpenIDConfiguration = errorOpenIDConfiguration;
        this.token = null;
    }
}
```

Summary of Configuration

The following table summarizes the proposed configuration for Kafka's SASL/OAUTHBEARER implementation.

Configuration	Example Value	Likelihood of Value Being Different from the Example Value	Notes

JAAS Login Module	`org.apache.kafka.common.security.oauthbearer.`
	
	`OAuthBearerLoginModule`
		Low	

Non-Broker:
 	
`sasl.login.callback.handler.class`
 	
Broker:
 	
`listener.name.sasl_ssl.oauthbearer.`
 	
`sasl.login.callback.handler.class`
	`org.example.MyLoginCallbackHandler`	High	Default implementation creates an unsecured OAuth Bearer Token

`listener.name.sasl_ssl.oauthbearer.`
 	
`sasl.server.callback.handler.class`
	`org.example.MyValidatorCallbackHandler`	High	Default implementation validates an unsecured OAuth Bearer Token

JAAS Module Options	Varied	High	Used to configure the above sasl.login and sasl.server callback handlers.
producer/consumer/broker configs: `sasl.login.refresh.*`	Varied, though defaults are reasonable	Low	

Summary for Production Use

To use SASL/OAUTHBEARER in a production scenario it is necessary to write two separate callback handlers implementing `org.apache.kafka.common.security.auth.AuthenticateCallbackHandler`:

- A **login callback handler** that can retrieve an OAuth 2 bearer token from the token endpoint and wrap that token as an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerToken`. It must attempt to do this when it is asked to handle an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerTokenCallback`.
- A **SASL Server callback handler** that can validate an OAuth 2 bearer token compact serialization and convert that compact serialization to an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerToken`. It must attempt to do this when it is asked to

handle an instance of `org.apache.kafka.common.security.oauthbearer.OAuthBearerValidatorCallback`. Note that this may entail secure retrieval (and perhaps caching) of a public key along with digital signature verification if the token is a JSON Web Signature (JWS); validation of various token claims such as the '`iat`' (Issued At), '`nbf`' (Not Before), '`exp`' (Expiration Time), '`aud`' (Audience), and '`iss`' (Issuer) claims if the token is a JSON Web Token (JWT); and/or decryption if the token is encrypted as a JSON Web Encryption (JWE).

See KIP-255 OAuth Authentication via SASL/OAUTHBEARER#Summary of Configuration for details on how to declare these two classes to Kafka.

It is likely that the implementations of the above two callback handlers will leverage an open source JOSE (Javascript Object Signing and Encryption) library. See https://jwt.io/ for a list of many of the available libraries.

OAuth 2 is a flexible framework that allows different installations to do things differently, so the principal name in Kafka could come from any claim in a JWT. Most of the time it would come from the '`sub`' claim, but it could certainly come from another claim, or it could be only indirectly based on a claim value (maybe certain text would be trimmed or prefixed/suffixed). Because the OAuth 2 framework is flexible, we need to accommodate that flexibility – and the ability to plugin arbitrary implementations of the above two callback handler classes gives us the required flexibility. As an example, the SASL Server callback handler implementation could leverage an open source JOSE library to parse the compact serialization, retrieve the public key if it has not yet been retrieved, verify the digital signature, validate various token claims, and map the '`sub`' claim to the `OAuthBearerToken`'s principal name (which becomes the SASL authorization ID, which becomes the Kafka principal name). By writing the callback handler code we have complete flexibility to meet the requirements of any particular OAuth 2 installation.

The following references may be helpful:

URL	Description
https://tools.ietf.org/html/rfc6749	RFC 6749: The OAuth 2.0 Authorization Framework
https://tools.ietf.org/html/rfc6750	RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage
https://tools.ietf.org/html/rfc7519	RFC 7519: JSON Web Token (JWT)
https://tools.ietf.org/html/rfc7515	RFC 7515: JSON Web Signature (JWS)
https://tools.ietf.org/html/rfc7516	RFC 7516: JSON Web Encryption (JWE)
https://tools.ietf.org/html/rfc7628	RFC 7628: A Set of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth

Proposed Changes

Thanks to KIP-86: Configurable SASL callback handlers, no changes to existing public interfaces are required – all functionality represents additions rather than changes. The only changes to the existing *implementation* are to define appropriate default callback handler/login classes for SASL/OAUTHBEARER.

Compatibility, Deprecation, and Migration Plan

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

  None

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

  There is no change to existing behavior

Rejected Alternatives

Motivation

The idea of providing a toolkit of reusable JWT/JWS/JWE retrieval and validation functionality was discarded in favor of a single, simple unsecured JWS implementation. Anything beyond simple unsecured use cases requires significant functionality that is available via multiple open source libraries, and there is no need to duplicate that functionality here. It is also not desirable for the Kafka project to define a specific open source JWT/JWS/JWE library dependency; better to allow installations to use the library that they feel most comfortable with. This decision also allows the public-facing interface of this KIP to be considerably smaller than it would otherwise be.

Substitution Within Configuration Values

Flexible, substitution-aware configuration was originally proposed in an early draft of this KIP but was separated out into its own KIP-269 Substitution Within Configuration Values because it was thought that such functionality might be more broadly useful. That KIP did not gain traction in part because it was pointed out that most secret-related configuration could be done dynamically via the functionality provided by KIP-226 Dynamic Broker Configuration, which can help remove secrets from the configuration files themselves and is based on an event notification paradigm as opposed to the passive, pull-based one used by substitution. It was also pointed out that production implementations of SASL/OAUTHBEARER for Kafka as defined here would have to provide their own `AuthenticateCallbackHandler` implementations (as per KIP-86: Configurable SASL callback handlers), and those implementation can be as flexible as desired/required with respect to how they deal with secrets. In the end, this KIP has no dependency on substitution, and that KIP can live/die/resurrect – as the community sees fit – independently of this one.

Explicit Configuration of Token Refresh Class

We adjusted the logic in `SaslChannelBuilder` to automatically configure the `org.apache.kafka.common.security.oauthbearer.internal.OAuthBearerRefreshingLogin` class as the `Login` implementation by default for SASL/OAUTHBEARER use cases. We also decided not to unify the refresh logic required for both SASL/OAUTHBEARER and SASL/GSSAPI mechanisms as part of this KIP, though this unification could occur at some point in the future. The above-mentioned `OAuthBearerRefreshingLogin` class (which is NOT part of the public API) delegates to an underlying imlementation in the same internal package, and this delegation-based approach suggests a potential way forward with regard to unification, but unification is explicitly out of scope here. The chosen delegation design along with automatic configuration allows several classes to be moved to the internal package, which helps to minimize the public-facing API for this KIP and creates a better out-of-the-box experience.

Callback Handlers and Callbacks

We decided to leverage the standard JAAS Callback/CallbackHandler mechanism for communicating information between various components (specifically, between the SASL Client and the Login Module, between the Login Module and the Login/Token Retrieval mechanism, and between the SASL Server and the Token Validation mechanism). We had originally documented the need for just the last two (token retrieval and token validation), and the original proposal was to declare the class names as part of the JAAS configuration. The only real benefit to doing this was that the declaration of the classes and their configuration were co-located in the JAAS configuration. We decided the benefit of consistency was at least as valuable as any cost associated with separating the declaration from the configuration – and probably more valuable given that this is how configuration is done for other SASL mechanisms and is therefore not unfamiliar. We also now leverage callback handlers in all three places where it is supported, though one of them (SASL Client callback handler) has no need for explicit configuration in any known use case. Finally, we adjusted the logic in `SaslChannelBuilder` and `LoginManager` to automatically apply default `AuthenticateCallbackHandler` and `Login` implementations for SASL/OAUTHBEARER in the absence of explicit declarations, which simplifies the out-of-the-box experience for unsecured (i.e. development and testing) use cases.