KIP-768: Extend SASL/OAUTHBEARER with Support for OIDC

Status

Current state: *Approved for 3.1.0*

Discussion thread: here

JIRA: KAFKA-13202

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

Motivation

The convergence on the use of OAuth/OIDC for authorization and authentication of Internet-based services is well underway. Use of a standard set of technologies allows organizations to select from among a range of OAuth/OIDC-compatible providers instead of defining, developing, and managing identity, security, and policy infrastructure on their own. Organizations can communicate with these providers using standard protocols (defined in RFCs) using code written against mature open source libraries in popular programming languages. There is already a rich (and growing!) ecosystem of tools to integrate with the providers. Although it is not a trivial task of selecting a provider and implementing an OAuth-compliant suite of applications and services in an organization, these standards provide flexibility within that organization.

When it comes to Kafka, the work done via KIP-255 (OAuth Authentication via SASL/OAUTHBEARER) introduced a framework that allowed for integration with OAuth-compliant providers. With this framework in place, Kafka clients could now pass a *JWT* access token to a broker when initializing the connection as a means of authentication. This means Kafka can start to leverage these standards for authorization and authentication for which many current and prospective customers are asking.

However, the KIP-255 quickly notes that:

> *OAuth 2 is a flexible framework with multiple ways of doing the same thing.*

Because of the high degree of flexibility required by different organizations as well as differences in implementations by various OAuth providers, “supporting OAuth” isn't as straightforward as one might hope. While flexibility is one of OAuth's strengths it also presents one of its greatest challenges for adopters. Because of OAuth's flexibility, the organizational requirements, and the minor differences in provider implementation, the exact means and logic by which a JWT access token is retrieved and validated may vary on a case-by-case basis. In fact, a generalized, out-of-the-box OAuth implementation was specifically *not included* in KIP-255 because:

> *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.*

The implementation of KIP-255 provided a concrete example implementation that allowed clients to provide an *unsecured* JWT access token to the broker when initializing the connection only for–as the KIP states–”development scenarios.”

The KIP sums up this intentional missing functionality by directing organizations wishing to adopt OAuth within Kafka:

Production use cases will require writing an implementation of `AuthenticateCallbackHandler` that can handle an instance of `OAuthBearerTokenCallback`

So the exact implementation is left up to each organization to implement. To fill the gap in the Apache Kafka project, a handful of open source Java implementations have been provided by the community. These implementations can be included in the class path of a OAuth-enabled Kafka client and configured appropriately to achieve the goal.

This project is to provide a concrete implementation of the interfaces defined in KIP-255 to allow Kafka to connect to an Open ID identity provider for authentication and token retrieval. While KIP-255 provides an unsecured JWT example for development, this will fill in the gap and provide a production-grade implementation.

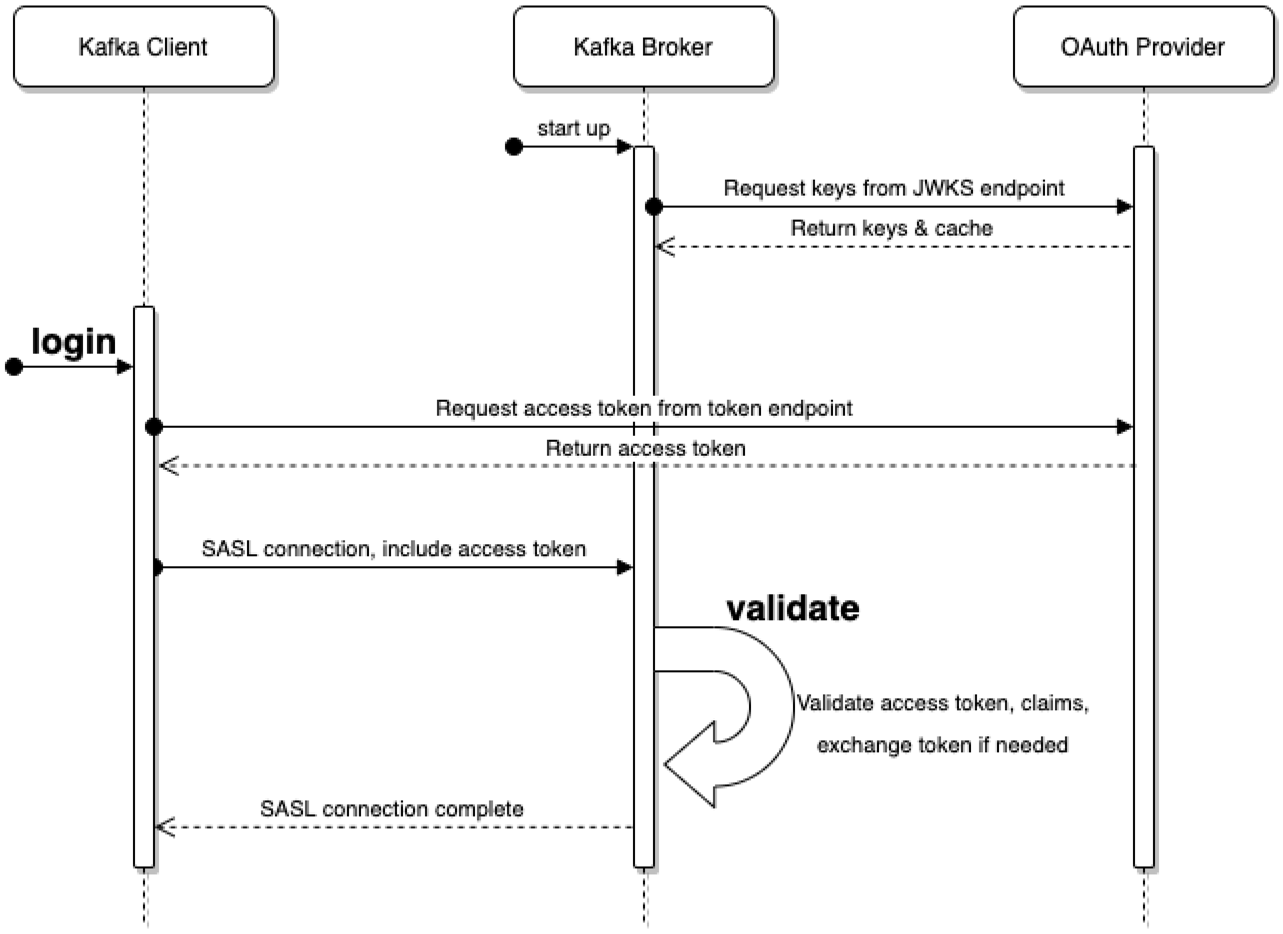
Public Interfaces

No changes to the public interface are anticipated; it will leverage the existing `AuthenticateCallbackHandler` API.

Proposed Changes

The OAuth/OIDC work will allow out-of-the-box configuration by any Apache Kafka users to connect to an external identity provider service (e.g. Okta, Auth0, Azure, etc.). The code will implement the standard OAuth `clientcredentials` grant type.

The proposed change is largely composed of a pair of `AuthenticateCallbackHandler` implementations: one to login on the client and one to validate on the broker.



As show in the above diagram, the login callback is executed on the client and the validate callback is executed on the broker.

Login Callback Handler (Client)

The login callback handler is invoked as part of the standard Kafka client login process. It is in this callback handler that the necessary HTTPS calls will be made to the OAuth provider using the provided configuration.

Token Retrieval Logic

Here is an example call to retrieve a JWT access token using `curl` and `jq`:

```
CLIENT_ID=abc123
CLIENT_SECRET='S3cr3t!'
URL=https://myidp.example.com/oauth2/default/v1/token
SCOPE=sales-pipeline

base_64_string=$(echo -n "$CLIENT_ID:$CLIENT_SECRET" | base64)

access_token=$(curl \
--silent \
--request POST \
--url $URL \
--header "accept: application/json" \
--header "authorization: Basic $base_64_string" \
--header "cache-control: no-cache" \
--header "content-type: application/x-www-form-urlencoded" \
--data "grant_type=client_credentials&scope=$SCOPE" | jq -r .access_token)
```

The OIDC specification requires the access token to be in the JWT format. The *client* will perform parsing and some basic structural validation of the JWT access token contents.

Client Configuration

The name of the implementation class will be `org.apache.kafka.common.security.oauthbearer.secured.OAuthBearerLoginCallbackHandler` and it will accept instances of `org.apache.kafka.common.security.oauthbearer.OAuthBearerTokenCallback` and `org.apache.kafka.common.security.auth.SaslExtensionsCallback`. The fully-qualified class name will be provided to the client's `sasl.login.callback.handler.class` configuration.

Because the HTTP call made to the OAuth/OIDC provider may time out or transiently fail, there will be a retry mechanism that waits between attempts. The number of attempts that are made (including the first attempt) are variable as it uses an exponential backoff approach; the first attempt to connect to the HTTP endpoint will be made immediately. If that first attempt fails, a second attempt will first wait a configurable number of milliseconds–`sasl.login.retry.backoff.ms`–before trying again. If that second attempt fails, the wait time will be doubled before a third attempt. This pattern repeats as needed up to the maximum wait time of of `sasl.login.retry.backoff.max.ms`.

There are several configuration options for this callback handler. Sensitive configuration options and SASL extensions appear under the JAAS configuration (`sasl.jaas.config`) while the rest are top-level configuration.

The JAAS configuration options are:

- `clientId`: supports OAuth `clientcredentials` grant type
- `clientSecret`: supports OAuth's `clientcredentials` grant
- `scope`: optional scope to reference in the call to the OAuth server

The top-level configuration options for the client login callback handler are:

- `sasl.oauthbearer.token.endpoint.url`: OAuth issuer token endpoint URL
- `sasl.oauthbearer.scope.claim.name`: optional override name of the scope claim; defaults to `scope`
- `sasl.oauthbearer.sub.claim.name`: optional override name of the sub claim; defaults to `sub`
- `sasl.login.connect.timeout.ms`: optional value in milliseconds for HTTPS connect timeout; defaults to `10000`
- `sasl.login.read.timeout.ms`: optional value in milliseconds for HTTPS read timeout; defaults to `10000`
- `sasl.login.retry.backoff.ms`: optional value in milliseconds for the amount of time to wait between HTTPS call attempts; defaults to `100`
- `sasl.login.retry.backoff.max.ms`: optional value in milliseconds for the *maximum* wait for HTTPS call attempts (as described above); defaults to `10000`

Here's an example of the configuration as a part of a Java properties file:

```
sasl.login.callback.handler.class=...OAuthBearerLoginCallbackHandler
sasl.login.connect.timeout.ms=15000
sasl.oauthbearer.token.endpoint.url=https://myidp.example.com/oauth2/default/v1/token
sasl.jaas.config=...OAuthBearerLoginModule required \
clientId="abc123" \
clientSecret="S3cr3t!" \
scope="sales-pipeline" \
extension_organizationId="sales-emea" ;
```

In the above example, the OAuth provider's `sasl.oauthbearer.token.endpoint.url` has been specified as well as an override of the default for `sasl.login.connect.timeout.ms`. The values for `clientId` and `clientSecret` as provided by the OAuth provider for an "API" or "machine-to-machine" account are required in the JAAS configuration. The optional `scope` value will allow the inclusion of a `scope` parameter when requesting the token.

Notice that there is also a SASL extension configuration in this example: `extension_organizationId`. Extensions will be ignored during the OAuth token retrieval step, but will be passed to the broker through the existing SASL extension mechanism from KIP-342.

Once the login has occurred for this client, the returned access token can be reused by other connections from this client. While these additional connections will not issue the token retrieval HTTP call on the client, the broker will still validate the token once for each time it is sent by each client connection.

Per KIP-368, the OAuth token re-authentication logic from the existing implementation is automatically "inherited" by this implementation, so no additional work is needed to support that feature.

Validator Callback Handler (Broker)

The validation callback handler is invoked as part of the SASL Kafka server authentication process when a JWT is received. This callback handler will parse and validate the JWT using its contents along with provided configuration.

Validation Logic

The basic overview of the JWT validation that will be performed is:

1. Parse the JWT into separate header, payload, and signature sections
2. Base-64 decode the header and payload
3. Extract the necessary claims for validation
4. Match the key ID (`kid`) specified in the JWT header to a JWK ID from the JWK Set
5. Ensure the encoding algorithm (`alg` from the header) isn't `none` and matches the expected algorithm for the JWK ID
6. Encode the header and payload sections of the original encoded JWT access token using the public key from the JWK and ensure it matches the signature section of the JWT
7. Extract the `scope`, `iat`, `exp`, and `sub` claims as these are needed by the `OAuthBearerToken` object to be passed to the `OAuthBearerValidatorCallback`.
8. Optional claim validation that ensures that issuer, audience, or other claims match a given value

The extensions validation will be executed the same as in `org.apache.kafka.common.security.oauthbearer.internals.unsecured.OAuthBearerUnsecuredValidatorCallbackHandler` today.

It's possible that a key ID (`kid`) could appear in the header of an incoming JWT access token that does not appear in the JWKS cached by a broker.

Broker Configuration

The name of the implementation class will be `org.apache.kafka.common.security.oauthbearer.secured.OAuthBearerValidatorCallbackHandler` and it will accept instances of `org.apache.kafka.common.security.oauthbearer.OAuthBearerValidatorCallback` and `org.apache.kafka.common.security.oauthbearer.OAuthBearerExtensionsValidatorCallback`. The fully-qualified class name will be provided to the broker's `listener.name.<listener name>.oauthbearer.sasl.server.callback.handler.class` configuration.

It may be that the names of the claims used by the OAuth provider differ from what is expected. For example, the security principal for which the token is valid is usually contained in the `sub` (subject) JWT claim. There may be cases where the value of that claim may not be valid or usable, and instead the value will need to be extracted from, for example, the `email` claim.

There are several configuration options for this callback handler. Since there are no sensitive configuration options, they are all in the top-level configuration. The configuration can be top-level or scoped to a specific listener with the listener prefix `listener.name.<listener name>.oauthbearer`. Here are the configuration options:

- `sasl.oauthbearer.jwks.endpoint.url`: OAuth issuer's JWK Set endpoint URL from which to retrieve the set of JWKs managed by the provider; this can be a `file://`-based URL that points to a broker file system-accessible file-based copy of the JWKS data. This allows the JWKS data to be updated on the file system and refreshed on the broker when the file is updated, thus avoiding any HTTP(S) communication with the OAuth/OIDC provider
- `sasl.oauthbearer.jwks.endpoint.refresh.interval.ms`: optional value in milliseconds for how often to refresh the JWKS from the URL pointed to by `sasl.oauthbearer.jwks.endpoint.url`. Only used when using an HTTP(S)-based URL for `sasl.oauthbearer.jwks.endpoint.url`. Defaults to `3600000` (1 hour)
- `sasl.oauthbearer.jwks.endpoint.retry.backoff.ms`: optional value in milliseconds for the amount of time to wait between HTTPS call attempts to retrieve the JWKS; only used when using an HTTP(S)-based URL for `sasl.oauthbearer.jwks.endpoint.url` ; defaults to 100
- `sasl.oauthbearer.jwks.endpoint.retry.backoff.max.ms`: optional value in milliseconds for the *maximum* wait for HTTPS call attempts to retrieve the JWKS; only used when using an HTTP(S)-based URL for `sasl.oauthbearer.jwks.endpoint.url` ; defaults to `1000`
- `sasl.oauthbearer.sub.claim.name`: name of the scope from which to extract the subject claim from the JWT; defaults to `sub`
- `sasl.oauthbearer.scope.claim.name`: name of the scope from which to extract the scope claim from the JWT; defaults to `scope`
- `sasl.oauthbearer.clock.skew.seconds`: optional value in seconds for the clock skew between the OAuth/OIDC provider and the broker. Only used when using an HTTP(S)-based URL for `sasl.oauthbearer.jwks.endpoint.url`. Defaults to `30`
- `sasl.oauthbearer.expected.audience`: The (optional) comma-delimited setting for the broker to use to verify that the JWT was issued for one of the expected audiences. The JWT will be inspected for the standard OAuth `aud` claim and if this configuration option is set, the broker will match the value from JWT's `aud` claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

- `sasl.oauthbearer.expected.issuer`: Optional setting for the broker to use to verify that the JWT was created by the expected issuer. The JWT will be inspected for the standard OAuth `iss` claim and if this configuration option is set, the broker will match the value from JWT's `iss` claim to see if there is an exact match. If there is no match, the broker will reject the JWT and authentication will fail.

Here's an example of the configuration as a part of a Java properties file:

```
listener.name.<listener name>.oauthbearer.sasl.server.callback.handler.class=...
OAuthBearerValidatorCallbackHandler
listener.name.<listener name>.oauthbearer.sasl.jaas.config=...OAuthBearerLoginModule required;
sasl.oauthbearer.jwks.endpoint.url=https://myidp.example.com/oauth2/default/v1/keys
sasl.oauthbearer.scope.claim.name=scp
```

In the above configuration the broker points to the appropriate OAuth provider `sasl.oauthbearer.jwks.endpoint.url` to retrieve a the set of JWKs for validation. In this example, a non-default value for `sasl.oauthbearer.scope.claim.name` has been provided because the provider uses `scp` for the name of the scope claim in the JWT it produces.

JWKS Management Logic

The JSON Web Key Set (JWKS) is a JSON document provided by the OAuth/OIDC provider that lists the keys used to sign the JWTs it issues.

Here is a sample JWKS JSON document:

```
{
  "keys": [
    {
      "kty": "RSA",
      "alg": "RS256",
      "kid": "abc123",
      "use": "sig",
      "e": "AQAB",
      "n": "..."
    },
    {
      "kty": "RSA",
      "alg": "RS256",
      "kid": "def456",
      "use": "sig",
      "e": "AQAB",
      "n": "..."
    }
  ]
}
```

Without going into too much detail, the array of `keys` enumerates the key data that the provider is using to sign the JWT signature. The key ID (`kid`) is referenced by the JWT's header in order to match up the JWT's signing key with the key in the JWKS. During the validation step, the jose4j OAuth library will use the contents of the appropriate key in the JWKS to validate the signature.

Given that the JWKS is referenced by the JWT, the JWKS must be made available by the OAuth/OIDC provider so that a JWT can be validated.

The JWKS will be kept up-to-date in two main ways:

1. Providing a JWKS URL. In this mode, the JWKS data will be retrieved from the OAuth provider via the configured URL on broker startup. All then-current keys will be cached on the broker (per the ‘max age’; the jose4j library has a means to keep these-up-to-date when they age out) for incoming requests. If an authentication request is received for a JWT that includes a `kid` that isn’t yet in the cache, the JWKS endpoint will be queried again on demand. However, we prefer polling via a background thread to hopefully pre-populate the cache with any forthcoming keys before any JWT requests that include them are received.
2. Providing a JWKS file. On startup, the broker will load the JWKS file from the configured location. Any updates to the file will require a restart of the broker. The means by which the JWKS file is updated is left to the cluster administrator. In the event that an unknown JWT key is encountered, this implementation will simply issue an error and validation will fail.

If the the URL or file that is specified cannot be read, the broker will fail to start up. In the case of an HTTP(S)-based URL, the configured `sasl.login.retry.backoff.ms` and `sasl.login.retry.backoff.max.ms` values will be used to make attempts to connect to the remote OAuth provider.

It is also important that the JWKS is retrieved before the broker's ports are opened. Otherwise clients that connect to the broker before the JWKS is retrieved will experience spurious authentication failures (e.g. during broker restarts).

It is expected that an OAuth provider will publish a new JWKS well in advance of issuing any JWTs that contain those keys. However, in the case that a broker receives a JWT with a key ID that it doesn't have stored in its cached JWKS, the broker will need to remediate the issue:

1. If the JWKS URL is HTTP(S)-based **and** if the broker hasn't already attempted to resolve the key ID, enqueue a background thread to reload the JWKS from the HTTP(S) endpoint. The broker will keep track of key ID resolution failures so it doesn't repeatedly attempt to do so. If the JWKS URL is `file://`-based, no remediation processing will occur.
2. Send an authentication failure to the client. Since it is unknown at this point in processing of the key ID is valid-but-missing or if the key ID is just invalid, the broker will always issue an authentication error. Between the authentication failure delay mechanism and any client retry, there may be sufficient time for the broker to update the JWKS.

The common case will be that the key ID is one that has been published and accessed recently. The broker shouldn’t need to reach out to the JWKS endpoint in the on-demand fashion described above normally.

> The broker will not perform any loading of the JWKS in the network thread. This needs to be performed either prior to opening the ports during startup (as described above) or performed in a background thread.

The code will need to have a means to expunge old JWKS that are no longer needed. If there is a tangible benefit, perhaps the broker can keep track of recently removed/aged out key IDs so as to provide a more helpful message to the user.

Broker-to-broker Support

The use of OAuth credentials for broker-to-broker communication will continue to be supported. As with the existing implementation, users can specify the protocols and implementations to use for broker-based communication. This would require providing the appropriate configuration for both client login and broker validation.

Testing

In addition to unit and integration tests, there will be a standalone tool in the `tools` directory/module named `org.apache.kafka.tools.OAuthCompatibilityTool`. This test can be run via the existing `bin/kafka-run-class.sh` script thusly:

```
./bin/kafka-run-class.sh org.apache.kafka.tools.OAuthCompatibilityTool \
  --client-id foo \
  --client-secret bar \
  --token-endpoint-url https://example.com/oauth2/v1/token \
  --jwks-endpoint-url https://example.com/oauth2/v1/keys
```

As seen in the example invocation above, the various configuration options for the client and broker properties are passed in via command line options. The test will connect to remote systems as needed to authenticate, retrieve tokens, retrieve JWKS, and perform validation (nothing is "mocked"). The command line options can be listed using the standard `--help` command line option.

This tool serves three basic purposes:

1. Allows validation that a given OAuth/OIDC provider is compliant with our implementation of the specification
2. Provides users a means to reproduce and diagnose issues for support needs
3. Provides a means to perform system tests against live providers for tracking regressions

The output of the test is easily consumable:

```
PASSED 1/5: client configuration
PASSED 2/5: client JWT retrieval
PASSED 3/5: client JWT validation
PASSED 4/5: broker configuration
PASSED 5/5: broker JWT validation
```

Additionally debugging can be selectively enabled using the standard `tools-log4j.properties` file if errors are detected by the tool. All configuration options supported by the client and server are also supported by the tool; it attempts to be as faithful as possible to the runtime logic to minimize discrepancies between the tool and the client and broker.

Compatibility, Deprecation, and Migration Plan

Users can continue to use the `OAuthBearerUnsecuredLoginCallbackHandler` and/or other `AuthenticateCallbackHandler` implementations. Users will need to update both clients and brokers in order to use the new functionality.

Rejected Alternatives

Removing `OAuthBearerUnsecuredLoginCallbackHandler`

Although this change will provide an out-of-the-box implementation of an `AuthenticateCallbackHandler` that can be used to communicate with OAuth /OIDC, the exist unsecured implementation is still usable for development and testing. Given that its non-secure status is in its name and the documentation, it shouldn’t need to be removed or deprecated at this time.

Standalone Plugin

Technically the new implementation could be developed and shipped as a plugin separate from the main Apache Kafka project. Community adoption would be improved by an in-tree solution. Including this inside Apache Kafka doesn’t preclude alternative `AuthenticateCallbackHandler` implementations from use by clients, if desired.