KIP-368: Allow SASL Connections to Periodically Re-Authenticate

Status

Current state: *Adopted (in 2.2)*

Discussion thread: *here*

JIRA: *KAFKA-7352*

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

Motivation

The adoption of KIP-255: OAuth Authentication via SASL/OAUTHBEARER in release 2.0.0 creates the possibility of using information in the bearer token to make authorization decisions. Unfortunately, however, Kafka connections are long-lived, so there is no ability to change the bearer token associated with a particular connection. Allowing SASL connections to periodically re-authenticate would resolve this. In addition to this motivation there are two others that are security-related. First, to eliminate access to Kafka for connected clients, the current requirement is to remove all authorizations (i.e. remove all ACLs). This is necessary because of the long-lived nature of the connections. It is operationally simpler to shut off access at the point of authentication, and with the release of KIP-86: Configurable SASL Callback Handlers it is going to become more and more likely that installations will authenticate users against external directories (e.g. via LDAP). The ability to stop Kafka access by simply disabling an account in an LDAP directory (for example) is desirable. The second motivating factor for re-authentication related to security is that the use of short-lived tokens is a common OAuth security recommendation, but issuing a short-lived token to a Kafka client (or a broker when OAUTHBEARER is the inter-broker protocol) currently has no benefit because once a client is connected to a broker the client is never challenged again and the connection may remain intact beyond the token expiration time (and may remain intact indefinitely under perfect circumstances). This KIP proposes adding the ability for SASL clients (and brokers when a SASL mechanism is the inter-broker protocol) to re-authenticate their connections to brokers and for brokers to close connections that continue to use expired sessions.

This KIP has no impact on non-SASL connections (e.g. connections that use the PLAINTEXT or SSL security protocols) – no such connection will be re-authenticated, and no such connection will be closed.

Public Interfaces

This KIP proposes the addition of a configuration option to enable the server-side expired-connection-kill feature (the option default results in no functionality change, of course, so there is no change to existing behavior in the absence of an explicit opt-in). This KIP also proposes bumping the version number for the SASL_AUTHENTICATE API to 1 (with a change in wire format) so that brokers can indicate the session expiration time to clients via an additional value on the last round-trip response. Clients can use the max SASL_AUTHENTICATE version number supported by the server to determine if they are connected to a broker that supports re-authentication (true if version > 0). This KIP also adds new metrics as described below.

The configuration option this KIP proposes to add to enable server-side expired-connection-kill is '**`connections.max.reauth.ms`**' (not required to be prefixed with listener prefix or SASL mechanism name – such a value would be used across the cluster – but it may be as mentioned below). For example, "`connections.max.reauth.ms=3600000`". The value represents the maximum value that could potentially be communicated as part of the new V1 SaslAuthenticateResponse. The default value is 0, which means there is effectively no maximum communicated (0 will be sent, meaning "none"), server-side kill of expired connections is disabled, clients are not required to re-authenticate, and whether clients re-authenticate or not and at what interval is entirely up to them. Existing SASL clients upgraded to v2.2.0 will be coded to not re-authenticate in this scenario. The default value of 0 therefore results in no change whatsoever.

When '**`connections.max.reauth.ms`**' is explicitly set to a positive number the server will disconnect any SASL connection that does not re-authenticate and subsequently uses the connection for any purpose other than re-authentication at any point beyond the communicated expiration point (which will not exceed the configured maximum value). For example, if the configured value is 3600000 (1 hour) and the remaining lifetime of a bearer token presented at the time of authentication is 45 minutes, then 45 minutes is communicated back to the client and the server would kill the connection if it is not re-authenticated within 45 minutes and it is then actively used for anything other than re-authentication. As a further example, if the configured value is 3600000 and the credential lifetime is either unspecified or greater than 1 hour then 1 hour would be communicated to the client and the server would kill the connection if it is not re-authenticated within 1 hour and it is then actively used for anything other than re-authentication.

Older clients will of course not have the session expiration time communicated to them since they will use a version 0 SaslAuthenticateRequest and will receive the existing version 0 SaslAuthenticateResponse. The broker will disconnect these connections upon session expiration regardless of the fact that the client is an older one. Such connections will be captured via a metric (described below) to help with migration.

The '**`connections.max.reauth.ms`**' configuration option will not be dynamically changeable; restarts will be required if the value is to be changed. However, if a new listener is dynamically added, the value could be set for that listener at that time (and the configuration key would be prefixed in that case); this dynamic capability will be addressed as a separate ticket and may not be delivered with the initial KIP implementation.

From a behavior perspective on the client side (including the broker when it is acting as an inter-broker client), when a v2.2.0-or-later SASL client connects to a v2.2.0 or later broker that supports re-authentication, the broker will communicate the session expiration time as part of the final SASL_AUTHENTICATE response. If this value is positive, then the client will automatically re-authenticate before anything else unrelated to re-authentication is sent beyond that expiration point. If the re-authentication attempt fails then the connection will be closed by the broker; retries are not supported. If re-authentication succeeds then any received responses that queued up during re-authentication along with the `Send` that triggered the re-authentication to occur in the first place will subsequently be able to flow through (back to the client and along to the broker, respectively), and eventually the connection will re-authenticate again, etc. Note also that the client cannot queue up additional send requests beyond the one that triggers re-authentication to occur until re-authentication succeeds and the triggering one is sent.

From a behavior perspective on the server (broker) side, when the expired-connection-kill feature is enabled with a positive value the broker will communicate a session time via SASL_AUTHENTICATE and will close a connection when the connection is used past the expiration time and the specific API request is not directly related to re-authentication (`SaslHandshakeRequest` and `SaslAuthenticateRequest`). In other words, if a connection sits idle, it will not be closed – something unrelated to re-authentication must traverse the connection before a disconnect will occur.

Metrics documenting re-authentications will be maintained in the `Selector` code. SOme will mirror existing metrics that document authentications. For example: `failed-reauthentication-{rate,total}` and `successful-reauthentication-{rate,total}`. There will also be separate `successful-authentication-no-reauth-{rate,total}` metrics to indicate the subset of clients that successfully authenticate with a V0 `SaslAuthenticateRequest` (or no such request, which can happen with very old clients) These metrics are helpful during migration (see Migration Plan for details) as they will identify if/when all clients are properly upgraded before server-side expired-connection-kill functionality is enabled: the rate metric will be zero across all brokers when it is appropriate to enable the feature, and the total metric will be unchanging (zero after a restart). There will also be `reauthentication-latency-{avg,max}` metrics that document the latency imposed by re-authentication. It is unclear if this latency will be problematic, and the data collected via these metrics may be useful as we consider this issue in the future.

An additional metric `ExpiredConnectionsKilledCount` will be created and maintained by the server-side `Processor` code to count the number of such events. It should remain at zero if all clients and brokers are upgraded to v2.2.0 or later. If it is non-zero then either an older client is connecting (which would be evidenced in the `successful-authentication-no-reauth` metrics mentioned above) or a newer client is not re-authenticating correctly (which would indicate a bug).

Proposed Changes

Implementation Overview

The description of this KIP is actually quite straightforward from a behavior perspective – turn the feature on with the configuration option in the broker and it just works. From an implementation perspective, though, the KIP is not so straightforward; a description of how it works therefore follows below. Note that this description applies to the implementation only – none of this is part of the public API.

This implementation works at a very low level in the Kafka stack, at the level of the network code. It is therefore transparent to all clients – it just works with no knowledge or accommodation required on their part. When a client makes a request to a broker the request is intercepted at the level of the `Selector` class and a check is done to see if re-authentication is enabled; if it is, and the broker supports re-authentication, then the connection is re-authenticated at that point before the request is allowed to flow through. The solution is elegant because it re-uses existing code paths while requiring no code changes higher up in the stack.

This KIP transparently adds re-authentication support for all uses, which at this point includes the following:

- `org.apache.kafka.clients.consumer.internals.ConsumerNetworkClient`
  - `org.apache.kafka.clients.consumer.KafkaConsumer`
  - `org.apache.kafka.connect.runtime.distributed.WorkerGroupMember`
- `kafka.controller.ControllerChannelManager`
- `org.apache.kafka.clients.admin.KafkaAdminClient`
- `org.apache.kafka.clients.producer.KafkaProducer`
- `kafka.coordinator.transaction.TransactionMarkerChannelManager`
- `kafka.server.ReplicaFetcherBlockingSend (kafka.server.ReplicaFetcherThread)`
- `kafka.admin.AdminClient`
- `kafka.tools.ReplicaVerificationTool`
- `kafka.server.KafkaServer`
- `org.apache.kafka.trogdor.workload.ConnectionStressWorker`

The final issue to describe is how/when a `KafkaChannel` instance (each of which corresponds to a unique network connection) is told to re-authenticate. Each `KafkaChannel` instance will remember the session expiration time communicated during (re-)authentication (if any); the code in the `Selector` class will check to see if that time has passed and will start the re-authentication process if so.

Compatibility, Deprecation, and Migration Plan

With respect to compatibility, there is no impact to existing installations because the default is for the server-side connection kill feature to be turned off, older clients never try to re-authenticate because they don't support it, and newer clients that connect to older brokers will know that the broker does not support re-authentication and will therefore not attempt it.

With respect to migration, the approach would be as follows:

1. Upgrade all brokers to v2.2.0 or later at whatever rate is desired with '**`connections.max.reauth.ms`**' allowed to default to 0. If SASL is used for the inter-broker protocol then brokers will check the SASL_AUTHENTICATE API version and use a V1 request when communicating to a broker that has been upgraded to 2.2.0, but the client will see the "0" session max lifetime and will not re-authenticate. Their connections will not be killed.
2. In parallel with (1) above, upgrade non-broker clients to v2.2.0 or later at whatever rate is desired. SASL clients will check the SASL_AUTHENTICATE API version and use a V1 request when communicating to a broker that has been upgraded to 2.2.0, but the client will see the "0" session max lifetime and will not re-authenticate. Their connections will not be killed.
3. After (1) and (2) are complete, perform a rolling restart of all brokers and check the metrics `successful-authentication-no-reauth-{rate,total}` to confirm that they remain at zero. This gives confidence that (1) and (2) are indeed complete.
4. Update '**`connections.max.reauth.ms`**' to a positive value and perform a rolling restart of brokers again.
5. Monitor the `successful-authentication-no-reauth-{rate,total}` metrics – they will remain at 0 unless an older client connects to the broker.

Rejected Alternatives

Delaying Support for Brokers Killing Connections

It was initially proposed that we defer adding the ability for brokers to kill connections using expired credentials to a future KIP. This functionality is actually easier to add than re-authentication, and re-authentication without this feature doesn't really improve security (because it can't be enforced). Adding the ability to kill connections using an expired bearer token without the ability for the client to re-authenticate also does not make sense as a general feature – it forces the client to essentially "recover" from what looks like a network error on a periodic basis. So we will implement both features at the same time.

Delaying Support for non-OAUTHBEARER SASL Mechanisms

It was initially proposed that we defer adding the ability for SASL clients to re-authenticate when using a non-OAUTHBEARER mechanism (e.g. PLAIN, GSSAPI, and SCRAM-related). We were able to identify how all mechanisms could be readily and easily supported.

Creating a Configuration Option to Disable Client-side Re-Authentication

It was initially proposed that we would create a client-side configuration option to disable the use of re-authentication on the client. This may have been necessary when we were contemplating not including support for non-OAUTHBEARER SASL mechanisms and/or when we had not decided to bump the SASL_AUTHENTICATE version number, but it because unnecessary given these decisions. There is no need to disable the feature if the client and the server both support it.

Validating the Token Lifetime as Part of Re-Authentication

It was initially proposed that we would have the broker reject (re-)authentications that occurred with a credential having a lifetime longer than the maximum allowed. This was decided to be unnecessary here because the same thing can be done as part of token validation.

Highest-Level Approach: Inserting Requests into Clients' Queues

The original implementation injected requests directly into both asynchronous and synchronous I/O clients' queues. This implementation was too high up in the stack and required significantly more work and code than any other solution. It also was much harder to maintain because it became entwined in the implementation of every client.

High-Level Approach: One-Size-Fits-All With Additional KafkaClient Methods

One implementation injected requests into the `KafkaClient` via a new method (shown below). This one-size-fits-all approach worked as long as synchronous I/O clients periodically checked for and sent injected requests related to re-authentication (via a new method, also shown below). This implementation was harder to maintain than the chosen approach because it crossed existing module boundaries related to security code, networking code, and code higher up in the stack – it imposed requirements (however slight) on code higher up in the stack in order for re-authentication to work. This violation of existing modularity caused concern. The code was also 2-3 times bigger (at least) relative to the accepted implementation, and it was incrementally (though not dramatically) more difficult to test. It did create the possibility of interleaving requests related to re-authentication with requests that the client was otherwise sending, which minimized latency spikes due to re-authentication, but that advantage was difficult to quantify and therefore did not tip the balance in favor of this option.

org.apache.kafka.clients.KafkaClient additions

```
    /**
     * Return true if any node has a re-authentication request either enqueued and
     * waiting to be sent or already in-flight. A call to {@link #poll(long, long)}
     * is required to send and receive/process the results of such requests. <b>An
     * owner of this instance that does not implement a run loop to repeatedly call
     * {@link #poll(long, long)} but instead only sends requests synchronously
     * on-demand to a single node must call this method periodically -- and invoke
     * {@link #poll(long, long)} if the return value is {@code true} -- to ensure
     * that any re-authentication requests that have been injected are sent and
     * processed in a timely fashion.</b>
     * <p>
     * Example code to re-authenticate a connection across several
     * requests/responses is as follows:
     *
     * <pre>
     * // Send multiple requests related to re-authentication in the synchronous
     * // use case, completing the re-authentication exchange.
     * while (kafkaClient.hasReauthenticationRequest())
     *     // Returns an empty list in synchronous use case.
     *     kafkaClient.poll(Long.MAX_VALUE, time.milliseconds());
     * // The connection is ready for use, and if there originally was a
     * // re-authentication request then as many requests as required to
     * // complete the exchange have been sent.
     * </pre>
     *
     * Alternatively, to only send one re-authentication request and receive its
     * response (which allows us to interleave other requests to the single node to
     * which we are connected before subsequent requests related to the multi-step
     * re-authentication exchange are sent):
     *
     * <pre>
     * // Send a single request related to re-authentication in the synchronous
     * // use case, potentially (but not necessarily) completing the
     * // re-authentication exchange.
     * while (kafkaClient.hasReauthenticationRequest()) {
     *     // Returns an empty list in synchronous use case.
     *     kafkaClient.poll(Long.MAX_VALUE, time.milliseconds());
     *     if (!kafkaClient.hasInFlightRequests())
     *         break; // Response has been received.
     * }
     * // The connection is ready for use, and if there was a
     * // re-authentication request then either the exchange is finished or
     * // there is another re-authentication request available to be sent.
     * </pre>
     *
     * @return if any node has a re-authentication request either enqueued and
     *         waiting to be sent or already in-flight
     * @see #enqueueAuthenticationRequest(ClientRequest)
     */
    default boolean hasReauthenticationRequest() {
        return false;
    }


    /**
     * Enqueue the given request related to re-authenticating a connection. This
     * method is guaranteed to be thread-safe even if the class implementing this
     * interface is generally not.
     *
     * @param clientRequest
     *            the request to enqueue
     * @see #hasReauthenticationRequest()
     */
    default void enqueueAuthenticationRequest(ClientRequest clientRequest) {
        // empty
    }
```

Adding an ExpiringCredential Public API

It was initially proposed that we make an existing, non-public `ExpiringCredential` interface part of the public API and leverage the background login refresh thread's refresh event to kick-start re-authentication on the client side for the refreshed credential. This is unnecessary due to a couple of factors. First, the server (broker) indicates to the client what the expiration time is, and the low-level mechanism we have chosen on the client side can insert itself into the flow at the correct time – it does not need an external mechanism; and second, the server will chose the token expiration time as the session expiration time if does not exceed the maximum allowable value, which means the refresh thread on the client side will have already refreshed the token (or, if it hasn't, the client can't make new connections anyway). We had at one time considered that the server rejecting tokens whose remaining lifetime exceeds the maximum allowable session time was a third factor, but that functionality was rejected because it can be done as part of token validation as mentioned above.

Authenticating a Separate Connection and Transferring Credentials

One alternative idea is to add two new request types: "ReceiveReauthenticationNonceRequest" and "ReauthenticateWithNonceRequest". When re-authentication needs to occur the client would make a separate, new connection to the broker and send a "ReceiveReauthenticationNonceRequest" to the broker to have it associate a nonce with the authenticated credentials and return the nonce to the client. Then the client would send a "ReauthenticateWithNonceRequest" with the returned nonce over the connection that it wishes to re-authenticate; the broker would then replace the credentials on that connection with the credentials it had previously associated with the nonce. I don't know if this would work (might there be some issue with advertised vs. actual addresses and maybe the possibility of there being a load balancer? Could we be guaranteed the ability to connect to the exact same broker as our existing connection?) . If it could work then it does have the advantage of requiring the injection of just a single request over an existing connection that would return very quickly rather than 3 separate requests of which at least one might take a while to return (to potentially retrieve a public key for token signature validation, for example; the validation itself isn't exactly free, either, even if the public key is already cached). One disadvantage of the alternative, nonce-based approach is that it requires the creation of a separate connection, including TLS negotiation, and that is very expensive compared to sending 3 requests over an existing connection (which of course already has TLS negotiated).

Brute-Force Client-Side Kill

A brute-force alternative is to simply kill the connection on the client side when the background login thread refreshes the credential. The advantage is that we don't need a code path for re-authentication – the client simply connects again to replace the connection that was killed. There are many disadvantages, though. The approach is harsh – having connections pulled out from underneath the client will introduce latency while the client reconnects; it introduces non-trivial resource utilization on both the client and server as TLS is renegotiated; and it forces the client to periodically "recover" from what essentially looks like a failure scenario. While these are significant disadvantages, the most significant disadvantage of all is that killing connections on the client side adds no security – trusting the client to kill its connection in a timely fashion is a blind and unjustifiable trust.

Brute-Force Server-Side Kill

We could kill the connection from the server side instead, when the token expires. But in this case, if there is no ability for the client to re-authenticate to avoid the killing of the connection in the first place, then we still have all of the harsh approach disadvantages mentioned above.