KIP-189: Improve principal builder interface and add support for SASL

- Status
- Motivation
- Proposed Changes
- Kafka Principal Semantics
- Rejected Alternatives

Status

Current state: *Adopted*

Discussion thread: here

JIRA: Unable to render Jira issues macro, execution error.

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

Motivation

Kafka exposes a `PrincipalBuilder` interface which can be used to derive custom principals in the context of SSL client authentication. Its current interface is this:

```
interface PrincipalBuilder extends Configurable {
  void configure(Map<String, ?> configs);

  Principal buildPrincipal(TransportLayer transportLayer, Authenticator authenticator) throws KafkaException;

  void close() throws KafkaException;
}
```

At the moment, the `Principal` derived from the `PrincipalBuilder` is converted to a `KafkaPrincipal` before being passed to the `Authorizer`. The `KafkaPrincipal` object is distinguished primarily by the fact that it has a principal type, which is always set to "User" when converting from a `Principal`. The `Authorizer` implementation depends on `KafkaPrincipal` indirectly through the following objects:

```
case class Session(principal: KafkaPrincipal, clientAddress: InetAddress)

case class Acl(principal: KafkaPrincipal, permissionType: PermissionType, host: String, operation: Operation)
```

In this KIP, we aim to solve two primary problems with the current `PrincipalBuilder` interface:

1. The `PrincipalBuilder` is currently only used for SSL authentication. We want to extend it to SASL mechanisms in general including GSSAPI. This includes unifying the Kerberos name translation rules.
2. Due to the conversion from `Principal` to `KafkaPrincipal`, a custom `Authorizer` implementation cannot leverage any enrichment provided at the authentication layer in a convenient way. For example, if the authorizer had a notion of groups, it might be reasonable to derive a group id from the client certificate OU field. In that case, the principal builder would have to pack that group id into the principal name to pass it through to the `Authorizer`. This might be reasonable for just one additional field, but it would be more general and much more convenient to pass through the enriched `Principal` all the way to `Authorizer`. (Note this is the problem which was trying to be solved in KIP-111.)

Additionally, the interface has a couple shortcomings from an API perspective that we want to address:

1. There is no use case that we are aware of which requires access to the `Authenticator` directly. The closest use case would be the SASL authenticator, but what we actually need is the `SaslServer`. Furthermore, there is an odd circular dependence between the `PrincipalBuilder` and the `Authenticator`: `Authenticator` exposes a `principal()` method which uses the `PrincipalBuilder` by passing itself as the second parameter.

2. There is no use case that we are aware of which requires the `TransportLayer` itself. What client SSL authentication actually needs is access to the `SSLSession` which is provided by the JRE. Hence we are unnecessarily exposing Kafka internals.

Proposed Changes

To address these problems, we propose first to introduce a new `AuthenticationContext` interface to encapsulate the authentication state needed to derive the principal. Initially we expose methods to get the underlying security protocol in use and the client address.

```
interface AuthenticationContext {
  String securityProtocolName();
  InetAddress clientAddress();
}
```

There will be two implementations of this interface exposed: `SslAuthenticationContext` and `SaslAuthenticationContext`. These expose the respective state needed to derive the `Principal`.

```
class SslAuthenticationContext implements AuthenticationContext {
  public final SSLSession session;
}

class SaslAuthenticationContext implements AuthenticationContext {
  public final SaslServer server;
}
```

Then we introduce a new builder interface to leverage the `AuthenticationContext`.

```
interface KafkaPrincipalBuilder {
  KafkaPrincipal build(AuthenticationContext context);
}
```

Instead of using the Java-provided `Principal` object, however, we return `KafkaPrincipal`. There are two reasons to use the `KafkaPrincipal` object instead of `Principal`. First, it guarantees that it can be passed through to the `Authorizer` without conversion which solves the KIP-111 problem. Second, it gives us an extension point for future usage inside Kafka. Let us illustrate with two examples:

1. With an enriched `Principal` object, it is a natural extension of the work on secure quotas to allow for quota enforcement at different granularities (e.g. by group). One way to do this would be to provide a `quotaPrincipal()` method in `KafkaPrincipal` which uses itself as the default implementation. Custom `PrincipalBuilder` implementations can extend `KafkaPrincipal` to expose the desired granularity of quota enforcement.
2. In the future, we may add support for groups to Kafka. This was brought up in the KIP-111 discussion. To support this, we can provide a `groupId()` method in `KafkaPrincipal` which defaults to a null value or an empty string. Extensions can override this just as before. Also note that it is still possible for the `Authorizer` implementation to derive its own group information for enforcement.

It would also be possible to return `Principal` and use an `instanceof` check for `KafkaPrincipal`, but we felt it is cleaner to require the `KafkaPrincipal` directly so that the principal type must be explicitly configured.

Kerberos name translation: Kafka exposes a convenience API for translating Kerberos authentication names into "short names." Basically users are allowed to configure a list of translation rules which are applied during the authentication process in order to derive the principal name. Currently this is implemented in `KerberosShortNamer` and is applied by the SASL callback handler when an `AuthorizeCallback` is received:

```
    private void handleAuthorizeCallback(AuthorizeCallback ac) {
        String authenticationID = ac.getAuthenticationID();
        ac.setAuthorized(true);
        KerberosName kerberosName = KerberosName.parse(authenticationID);
        String userName = kerberosShortNamer.shortName(kerberosName);
        ac.setAuthorizedID(userName);
    }
```

Since the `AuthorizeCallback` is not necessarily tied to Kerberos authentication, we propose to move this translation logic into a default implementation of `KafkaPrincipalBuilder`. If the authentication context is a `SaslAuthenticationContext`, then we check the mechanism and if it is "GSSAPI," we apply the short name rules. From a high level, the default `KafkaPrincipalBuilder` will look something like this:

```
class DefaultPrincipalBuilder {
  KafkaPrincipal build(AuthenticationContext context) {
    if (context instanceof SaslAuthenticationContext) {
      SaslAuthenticationContext saslContext = (SaslAuthenticationContext) context;
      if (saslContext.server.getMechanismName().equals("GSSAPI") {
        // apply kerberos short name rules
      } else {
        // return user principal derived from SaslServer.getAuthorizationID()
      }
    } else if (context instanceof SslAuthenticationContext) {
      // apply old principal builder if one was defined
      // otherwise pass through SSLSession.getPeerPrincipal
    } else {
      // throw some error for unexpected authentication types
    }
  }
}
```

Finally, we change the default handling of the `AuthorizeCallback` to simply pass through the authenticationId.

```
    private void handleAuthorizeCallback(AuthorizeCallback ac) {
        String authenticationID = ac.getAuthenticationID();
        ac.setAuthorized(true);
        ac.setAuthorizedID(authenticationID);
    }
```

In future work, a similar naming translation mechanism could be added to build principals from a certificate distinguished name.

Kafka Principal Semantics

A principal in Kafka is anything which can be granted permissions. Each principal is identified by a principal type and a name. So what does the enrichment that we are providing actually represent? For example, are they additional attributes used to identify the principal? If so, then the ACL command line tool must take these attributes into account when defining ACLs. We take an alternative view. Specifically:

1. A principal is always identifiable by a principal type and a name. Nothing else should ever be required.
2. Principal enrichment during authentication is merely a way to represent relations between the authenticated principal and other principals (possibly of a different type).
3. An authorizer may or may not take these relations into account when enforcing ACLs. It is valid to ignore them and treat the principal only as a user.

An example will make this clearer. A user may implement an authorizer which supports group ACLs. The group could be passed through the authentication layer (say if it derived from a client certificate) in a custom `KafkaPrincipal`.

```
class UserPrincipalAndGroup extends KafkaPrincipal {
  private final String userId;
  private final String groupId;

  public UserPrincipalAndGroup(String userId, String groupId) {
    super(KafkaPrincipal.USER_TYPE, userId);
    this.groupId = groupId;
  }

  public KafkaPrincipal group() {
    return new KafkaPrincipal(KafkaPrincipal.GROUP_TYPE, groupId);
  }
}
```

In this example, there are two principal types: user and group. The `UserPrincipalAndGroup` represents the user principal and its relation to a specific group principal. When used in the context of the `SimpleAclAuthorizer`, the group information is disregarded since this authorizer is not aware of groups. However, a group-aware authorizer could check ACLs for the corresponding group. The advantage of this approach is that it allows the authorizer to be agnostic of how the group is derived.

Note that the ACL command line tool is sufficient for this purpose. For example, we might create an ACL for a specific group using a command like this:

```
$ bin/kafka-acls --authorizer GroupAuthorizer --add --allow-principal Group:test-group --producer --topic Test-topic
```

The command line tool is sufficient as long as ACLs only take into account simple principals. Extensions would be needed to create ACLs based on relations between principals. This is outside the scope of this work.

Compatibility, Deprecation, and Migration Plan

We intend to deprecate and eventually remove the old `PrincipalBuilder` interface. For now it will continue to be supported in its current usage. We will not support SASL authentication through the `PrincipalBuilder` interface.

Both `PrincipalBuilder` and `KafkaPrincipalBuilder` will be exposed through the `principal.builder.class` configuration.
To avoid confusion when using extensions of `KafkaPrincipal`, we intend to deprecate and eventually remove the static `fromString` method since it only supports construction of `KafkaPrincipal` instances.

Rejected Alternatives

Another option to add support for SASL might be to modify the `SaslServerAuthenticator` to use the existing `PrincipalBuilder`. This allows us to write a custom `PrincipalBuilder` such as the following:

```
abstract class SaslPrincipalBuilder() implements PrincipalBuilder {

   Principal buildPrincipal(TransportLayer transportLayer, Authenticator authenticator) throws KafkaException {
       SaslServerAuthenticator saslAuthenticator = (SaslServerAuthenticator) authenticator;
       return buildPrincipal(saslAuthenticator.saslServer());
   }

   abstract Principal buildPrincipal(SaslServer server);
}
```

This was rejected primarily because it does nothing to clean up the current messy `PrincipalBuilder` API which leaks internal abstractions unnecessarily.

We also considered exposing the authentication context through the builder itself:

```
public abstract class KafkaPrincipalBuilder {
  public KafkaPrincipalBuilder setSaslServer(SaslServer server) {
    this.saslServer = saslServer;
    return this;
  }

  public KafkaPrincipalBuilder setSslSession(SslSession session) {
    this.session = session;
    return this;
  }

  public abstract KafkaPrincipal build();
}
```

This achieves the same goals and avoids the need for casting of the `AuthenticationContext` object. On the other hand, the implementation would have to rely on `null` checks to determine which case should be handled so it's a bit of a wash from that perspective. We preferred the proposed approach because we felt the contract was cleaner when all the necessary authentication state was passed directly through the `build` method rather than indirectly through the internal state of the builder itself. Admittedly, this is a subjective call.