KIP-342: Add support for Custom SASL extensions in OAuthBearer authentication

Status

Current state: *"Accepted"*

Discussion thread: *here*

JIRA: *here*

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

Motivation

Kafka currently supports non-configurable SASL extensions in its SCRAM authentication protocol for delegation token validation.

It would be useful to provide configurable SASL extensions for the OAuthBearer authentication mechanism as well, such that clients could attach arbitrary data for the principal authenticating into Kafka. Even though the JWT token standard supports customizable fields (in the form of claims), there are use cases where the client is unable to add additional ones (e.g: Kafka client receives a signed JWT token from a third-party).

This way, a custom principal can hold information derived from the authentication mechanism, which could prove useful for better tracing and troubleshooting, for example. This can be done in a way which allows for easier extendability in future SASL mechanisms.

It is worth noting that these extensions would lack a digital signature and therefore should not be used for critical use-cases where security is a concern.

Public Interfaces

New JAAS config option for default, unsecured bearer tokens - `unsecuredLoginExtension_<extensionname>` (as shown in the "Example" paragraph).
The name "auth" is not supported as a custom extension name with any SASL/OAUTHBEARER mechanism, including the unsecured one, since it is reserved by the specification for what is normally sent in the HTTP Authorization header. An attempt to use it will result in an exception on the client. There are also additional regex validations for extension name and values to ensure they conform to the SASL/OAUTHBEARER standard (specifically, https://tools.ietf.org/html/rfc7628#section-3.1)
The server can further validate the extensions via its pluggable callback handler.

`OAuthBearerExtensionsValidatorCallback` - callback for OAuth extension validation, providing access to the token

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

/**
 * A {@code Callback} for use by the {@code SaslServer} implementation when it
 * needs to validate the SASL extensions for the OAUTHBEARER mechanism
 * Callback handlers should use the {@link #validate(String)}
 * method to communicate valid extensions back to the SASL server.
 * Callback handlers should use the
 * {@link #error(String, String)} method to communicate validation errors back to
 * the SASL Server.
 * As per RFC-7628 (https://tools.ietf.org/html/rfc7628#section-3.1), unknown extensions must be ignored by the
server.
 * The callback handler implementation should simply ignore unknown extensions,
 * not calling {@link #error(String, String)} nor {@link #validate(String)}.
 * Callback handlers should communicate other problems by raising an {@code IOException}.
 * <p>
 * The OAuth bearer token is provided in the callback for better context in extension validation.
 * It is very important that token validation is done in its own {@link OAuthBearerValidatorCallback}
 * irregardless of provided extensions, as they are inherently insecure.
 */
public class OAuthBearerExtensionsValidatorCallback implements Callback {
    public OAuthBearerExtensionsValidatorCallback(OAuthBearerToken token, SaslExtensions extensions)

    /**
     * @return {@link OAuthBearerToken} the OAuth bearer token of the client
     */
    public OAuthBearerToken token()

    /**
     * @return {@link SaslExtensions} consisting of the unvalidated extension names and values that were sent
by the client
     */
    public SaslExtensions inputExtensions()

    /**
     * @return an unmodifiable {@link Map} consisting of the validated and recognized by the server extension
names and values
     */
    public Map<String, String> validatedExtensions()

    /**
     * @return An immutable {@link Map} consisting of the name->error messages of extensions which failed
validation
     */
    public Map<String, String> invalidExtensions()

    /**
     * Validates a specific extension in the original {@code inputExtensions} map
     * @param extensionName - the name of the extension which was validated
     */
    public void validate(String extensionName)

    /**
     * Set the error value for a specific extension key-value pair if validation has failed
     *
     * @param invalidExtensionName
     *            the mandatory extension name which caused the validation failure
     * @param errorMessage
     *            error message describing why the validation failed
     */
    public void error(String invalidExtensionName, String errorMessage)
}
```

`SaslExtensionsCallback` - generic callback to hold extensions

SaslExtensionsCallback

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


public class SaslExtensionsCallback implements Callback {
    /**
    * Returns a {@link SaslExtensions} consisting of the extension names and values that are sent by the client
to
    * the server in the initial client SASL authentication message.
    */
    public SaslExtensions extensions()

    /**
     * Sets the SASL extensions on this callback.
     */
    public void extensions(SaslExtensions extensions)
}
```

`SaslExtensions` - class for holding extensions data

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

/**
 * A simple value object class holding customizable SASL extensions
 */
public class SaslExtensions {
    public SaslExtensions(Map<String, String> extensionMap)

    /**
     * Returns an <strong>immutable</strong> map of the extension names and their values
     */
     public Map<String, String> map()
}
```

The default `OAuthBearerLoginModule` and the `OAuthBearerSaslClient` will be changed to request the extensions from their callback handler. For backwards compatibility it is not necessary for the callback handler to support `SaslExtensionsCallback`. Any UnsupportedCallbackException that is thrown will be ignored and no extensions will be added.

Proposed Changes

Describe the new thing you want to do in appropriate detail. This may be fairly extensive and have large subsections of its own. Or it may be a few sentences. Use judgement based on the scope of the change.

Create a new public `SaslExtensions` class that takes most of the generalizable logic from `ScramExtensions`. `ScramExtensions` will extend `SaslExtensions`
Create a new public `SaslExtensionsCallback` class which will be similar to `ScramExtensionsCallback`. `ScramExtensionsCallback` will **NOT** extend `SaslExtensionsCallback` since it will **not** support the new `SaslExtensions` class.
Create a new public `OAuthBearerExtensionsValidatorCallback` class.

Client Path

1. Pass `SaslExtensionsCallback` to the callback handler of `OAuthBearerLoginModule`. The handler should parses the extensions from the JAAS config (unsecuredLoginExtension_xxx) and populate them in the Subject class.
    a. The default `OAuthBearerUnsecuredLoginCallbackHandler` will be updated with this behavior.
2. Pass `SaslExtensionsCallback` to the callback handler of `OAuthBearerSaslClient`. The handler should take the extensions from the Subject and populate them in the callback
    a. The default `OAuthBearerSaslClientCallbackHandler` will be updated to handle the callback.
3. `OAuthBearerSaslClient` will then attach the populated extensions (if any) to the first client message

Server Path - `OAuthBearerServer`

1. Parse sent extensions (if any) from the first client message
    a. The OAuthBearerServer will use a strict regex which parses only letters for keys and only ASCII characters for values. This ensures the message conforms to the standard
2. Validate them by passing `OAuthBearerExtensionsValidatorCallback` to its callback handler

    a. If the configured server callback handler does not support `OAuthBearerExtensionsValidatorCallback`, no extensions will be exposed (as per RFC 7628 - *"Unknown key/value pairs MUST be ignored by the server"*)
3. Expose them via its `OAuthBearerServer#getNegotiatedProperty()` method.
    a. This will allow custom principals to access extensions through the `SaslServer` instance in `SaslAuthenticationContext#server()`

Example

A user would make use of the changes in this KIP in the following way:

1. Add the extension names to your JAAS configuration in the client

```
KafkaClient {
    org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule Required
    unsecuredLoginStringClaim_sub="thePrincipalName"
        unsecuredLoginExtension_traceId="123"
        unsecuredLoginExtension_logLevel="WARN";
};
```

2. A custom principal builder can then make use of the new extension

```
public class CustomPrincipalBuilder implements KafkaPrincipalBuilder {
    @Override
    public KafkaPrincipal build(AuthenticationContext context) {
        if (context instanceof SaslAuthenticationContext) {
            SaslServer saslServer = ((SaslAuthenticationContext) context).server();
            String traceId = saslServer.getNegotiatedPropery("traceId");
            return new CustomPrincipal("", saslServer, traceId);
        }
        throw new Exception();
    }
}
```

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users?*
  None. This simply allows for more configuration. We will ignore if legacy callback handlers raise `UnsupportedCallbackException` on the new callback classes.
  Other mechanisms like SCRAM should remain unaffected

Rejected Alternatives

- Add customizable extensions to every SASL client
    - | Not easily implementable, as we depend on a third-party library for PLAIN authentication
    - It is possible we implement configurable extensions for SCRAM as well. We would need to simply remove the checks in `ScramSaslServer` and then any custom callback handler could populate the extensions

As such, I decided it is best we limit the scope of this KIP while still implementing it in a way which would support future extensions by other SASL mechanisms