KIP-506: Allow setting SCRAM password via Admin interface

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *Under Discussion*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *KAFKA-8780*

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

Motivation

Adding support for setting SCRAM credentials via the Admin interface is beneficial:

- Having a Java API for handling user credentials is useful for external tooling which manages users.
- Currently it is only possible to create SCRAM credentials for users using the `kafka-configs.sh` script; there is no equivalent functionality in the Admin interface. But the script makes use of a direct ZooKeeper connection in order to create the znodes needed for storing user credentials. Since KIP-4 there has been a push to remove the need for ZooKeeper connections from scripts and replace them by the AdminClient API.

Setting credentials via the Admin interface necessitates two network hops:

- between the client and a broker
- between the broker and a zookeeper node

Security of ClientBroker communication

We must choose between transmitting the password itself or processing the password on the client and transmitting the salt, storedKey, serverKey and iterations (henceforth called "processed SCRAM information") to the broker.
The password itself is obviously security sensitive, so it would be insecure to transmit this to a broker without encryption.
While transmitting the processed SCRAM information wouldn't leak the password, it has other drawbacks:

- It permits an evesdropper to perform a brute force attack.
- Leaking the processed SCRAM information would break the mutual authentication property offered by SCRAM.
  Specifically, if an attacker can obtain the processed SCRAM information they could later perform a Man-in-the-Middle attack and the client would successfully but erronesouly authenticate the MitM as the broker.
- Since the password never arrives at the broker it makes it impossible for the broker to apply a complexity policy on the password in a centralized way.

This KIP lays out the protocol and API for setting SCRAM credentials via the Admin interface by transmitting
the **password** to the broker, which will perform the SCRAM processing and store that information in ZooKeeper.
As such by default it will only be possible to set SCRAM credentials over TLS connections, in order to not leak the password. A broker config is provided to allow administrators to enable password change over plain connections according to their trust in their network.
A future KIP may allow setting SCRAM credentials over plain connections.

This KIP also adds broker config to apply a password complexity policy and control which SCRAM mechanisms can be set via the Admin API.

Security of BrokerZookeeper communication

Kafka deployments often use non-TLS communication between broker and ZooKeeper. The `kafka-configs.sh` script can be run on a zookeeper node (communicating with it over localhost) which avoids sending processed SCRAM information over the network. It is, however, unclear how many admins actually do this.
With this KIP the processed SCRAM information would be sent unecrypted between the broker and ZooKeeper, which has all the drawbacks described in the previous section. However:

- Kafka and ZooKeeper nodes are often co-located on a secure network (this is not necessarily the case for clients and brokers)
- It is possible to deploy Kafka and ZooKeeper using TLS (for example, using more recent versions of ZooKeeper, or using a TLS tunnel)
- It is expected that the eventual removal of Kafka's dependency on ZooKeeper will remove this weakness.

Public Interfaces

- *New broker configs*
- *New message type in the protocol*
- *New methods in the Admin/AdminClient API*

Proposed Changes

Broker configs

Add new broker config **`sasl.scram.password.change.enabled`** will control whether the Admin API can be used for password change. It will take one of three possible values:

- `disabled` (the default) The Admin API cannot be used at all (the broker will return an error).
- `enabled_over_tls` The Admin API can only be used over a TLS connection.
- `enabled` The Admin API can be used over TLS and plain connections. This is only suitable for use where the client and broker are on the same trusted network.

Add new broker config **`sasl.scram.password.change.enabled.mechanisms`** to control the SASL mechanisms which can have passwords set via the Admin API.
This will not affect the existing `kafka-configs.sh` script. The default value will be `SHA256,SHA512`.

Add new broker config **`sasl.scram.password.change.<mechanism>.iterations`** to control the number of iterations to be used when SCRAM passwords are set. Defaults:

- sasl.scram.password.change.SHA256.iterations=4096
- sasl.scram.password.change.SHA512.iterations=4096

Add new broker **config `sasl.scram.password.policy.class`** to specify a password complexity policy.
Such classes will need to implement the Java interface:

```
        interface ScramPasswordPolicy {
            /**
             * Return whether the given password is acceptable according to the policy.
             * @return true if the given password is acceptable.
             */
            boolean accept(String password);

                /**
             * A human-readable description of the policy, which will be included in errors to the client.
             */
        String describe();
        }
```

By default no password policy will be applied.

New protocol

The client sends a `ScramPasswordRequest` containing the usernames, SCRAM hash types and passwords to be set. Note that the existing possibility of having a different password for different mechanisms is retained.

```
       {
         "type": "request",
         "name": "ScramPasswordRequest",
         "validVersions": "0",
         "fields": [
           { "name": "Credentials", "type": "[]ScramCredentials", "versions": "0+",
             "about": "The SCRAM credentials to set.",
             "fields": [
                   { "name": "Username", "type": "string", "versions": "0+",
                     "about": "The username." },
                 { "name": "HashType", "type": "int8", "versions": "0+",
             "about": "The scram hash type." },
               { "name": "Password", "type": "string", "versions": "0+",
                     "about": "The password." }
           ]}
         ]
       }
```

The broker then:

- The broker may reject the request, depending on the value of the **`sasl.scram.password.change.enabled`** config, otherwise
- The broker validates each password against any configured policy,
- The broker performs the SCRAM credential processing,
- The broker asynchronously updates ZooKeeper
- The broker returns `ScramPasswordResponse`

```
       {
         "type": "response",
         "name": "ScramPasswordResponse",
         "validVersions": "0",
         "fields": [
           { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
             "about": "The duration in milliseconds for which the request was throttled due to a quota
violation, or zero if the request did not violate any quota." },
           { "name": "HashResults", "type": "[]HashResult", "versions": "0+",
             "about": "Per-user results, grouped by hash type.",
             "fields": [
                   { "name": "HashType", "type": "int8", "versions": "0+",
                     "about": "The hash type." },
                   { "name": "UserResults", "type": "[]UserResult", "versions": "0+",
                     "about": "Per-user results for this hash type.",
                   "fields": [
                     { "name": "Username", "type": "string", "versions": "0+",
                       "about": "The username." },
                     { "name": "ErrorCode", "type": "int16", "versions": "0+",
                       "about": "The error code, or 0 if there was no error." },
                     { "name": "ErrorMessage", "type": "string", "nullableVersions": "0+", "versions": "0+",
                       "about": "The error message." }
                   ]}
             ]}
         ]
       }
```

Possible errors include:

- API_DISABLED (new code) The broker was configured with **`sasl.scram.password.change.enabled`**=disabled. The broker rejects the request and each user in the response will have error this error.
- `ENCRYPTION_REQUIRED` (new code) The request was made over a non-TLS connection but the broker was configured with **`sasl.scram.password.change.enabled`**=enabled_over_tls. The broker rejects the request and each user in the response will have error this error.
- `CLUSTER_AUTHORIZATION_FAILED` The authenticated user did not have Alter on the Cluster resource. In this KIP only users who can alter the cluster will be allowed to set SCRAM passwords, and therefore either all the users in the response will have this or none will. In the future an authorizer might support some non-super users being able to set the password of others, or users being able to set their own passwords.
- `POLICY_VIOLATION`: A given users password violated the password complexity policy. The message will explain the policy.
- `UNSUPPORTED_SASL_MECHANISM` A given users password was to be set for an unsupported Hash.

Admin API

The following method will be added to `Admin` and its subclasses:

```
        SetScramPasswordResult setScramPassword(Collection<ScramCredential> credentials)

        SetScramPasswordResult setScramPassword(Collection<ScramCredential> credentials,
SetScramPasswordRequestOptions options)
```

Where

```
        /**
         * A SCRAM entity identifies a user and a particular SCRAM hash type
         */
        class ScramEntity {
          enum Hash {
            SHA_256
            SHA_512
          }
          String userName;
          Hash hash;
          // equals() and hashCode()
        }

        /**
         * Combines a username, SCRAM hash type and a password
         */
        class ScramSecret {
          ScramEntity entity;
          String password;
        }

        class SetScramPasswordResult {
          KafkaFuture<Void> all();
          Map<ScramEntity, KafkaFuture<Void>> values();
        }
```

The implementation will:

1. Check whether the connection is via TLS and if not fail the request immediately.
2. Make the `ScramPasswordRequest`.

Setting a null password will delete the password from ZooKeeper.

Compatibility, Deprecation, and Migration Plan

- The `kafka-configs.sh` would continue to be supported with the `--zookeeper` option (it would not be deprecated yet, since this KIP does not yet provide equivalent functionality).

Rejected Alternatives

- Performing the SCRAM pre-processing on the client. See the drawbacks described in the motivation.

Future Work

- This KIP intentionally limits who can set passwords to the users with cluster admin. This is because the current `alterAcls()` operation in the `AdminClient` does not support a `User` `ResourceType. Support for ACLs on User resource can be implemented via a future KIP.
- This KIP only allows setting passwords over a TLS connection. A future KIP could enable setting passwords over plain connections, for example by using Diffie-Hellman key exchange to construct a shared secret to allow for encryption of the password over the client-broker connection.