Interface Design for Enhanced Authentication

1. Introduction to Enhanced Authentication (Referencing the MQTT 5 specification)

Enhanced authentication extends the basic authentication to include challenge/response style authentication. It might involve the exchange of **`AUTH`** packets between the Client and the Server after the **`CONNECT`** and before the **`CONNACK`** packets.

To begin an enhanced authentication, the Client includes an Authentication Method in the **`CONNECT`** packet. This specifies the authentication method to use. If the Server requires additional information to complete the authentication, it can send an **AUTH** packet to the Client. The Client responds to an **`AUTH`** packet from the Server by sending further **`AUTH`** packet. The Client and Server exchange **`AUTH`** packets as needed until the Server accepts the authentication by sending a **`CONNACK`** with Reason Code of 0.

1.1 Examples

- **Non-normative example showing a SCRAM challenge**

  Client to Server: CONNECT Authentication Method="SCRAM-SHA-1" Authentication Data=client-first-data

  Server to Client: AUTH rc=0x18 Authentication Method="SCRAM-SHA-1" Authentication Data=server-first-data ;

  Client to Server: AUTH rc=0x18 Authentication Method="SCRAM-SHA-1" Authentication Data=client-final-data

  Server to Client: CONNACK rc=0 Authentication Method="SCRAM-SHA-1" Authentication Data=server-final-data

- **Non-normative example showing a Kerberos challenge**

  Client to Server: CONNECT Authentication Method="GS2-KRB5"

  Server to Client: AUTH rc=0x18 Authentication Method="GS2-KRB5"

  Client to Server: AUTH rc=0x18 Authentication Method="GS2-KRB5" Authentication Data=initial context token

  Server to Client: AUTH rc=0x18 Authentication Method="GS2-KRB5" Authentication Data=reply context token

Client to Server: AUTH rc=0x18 Authentication Method="GS2-KRB5" - Server to Client: CONNACK rc=0 Authentication

1.2 Re-authentication

If the Client supplied an Authentication Method in the CONNECT packet it can initiate a re-authentication at any time after receiving a CONNACK. It does this by sending an AUTH packet with a Reason Code of 0x19 (Re-authentication).

2. User-Interface Design for Enhanced Authentication Support

2.1 Overview

Introducing the `AuthManager`, a trait that defines the interface for handling authentication packets. Modify `MqttOptions` module to include a new API called `set_auth_manager`, which allows users to set their own authentication manager that implements the `AuthManager`.

Additionally, modify the `Client` module to include a new API `reauth`, which enabled users to send an Auth packet for re-authentication purposes.

2.2 AuthManager Trait Design

The `AuthManger` is a trait that defines the interface for handling authentication packets. It has one method `auth_continue` for processing authentication data received from the server and generating authentication data to be sent back. This trait is designed to be implemented by the application to handle authentication flows.

The **`AuthManager`** is defined as follows:

```
pub trait AuthManager {
    /// Process authentication data received from the server and generate
authentication data to be sent back.
    ///
    /// # Arguments
    ///

    /// * `auth_prop` - The authentication Properties received from the server.
    ///
    /// # Returns
    ///
    /// * `Ok(auth_prop)` - The authentication Properties to be sent back to the
server.
    /// * `Err(error_message)` - An error indicating that the authentication process
has failed or terminated.
```

```
    fn auth_continue(&mut self, auth_prop: Option<AuthProperties>) -> Result<
Option<AuthProperties>, String>;
}
```

2.3 MqttOptions::set_auth_manager API Design

The `set_auth_manager` is an API that allows users to set their own authentication manager that implements the `AuthManager`, which is defined as follows:

```
pub fn set_auth_manager(&mut self, auth_manager: Arc<Mutex<dyn
AuthManager>>) -> &mut Self;
```

This method takes an `Arc<Mutex<dyn AuthManager>>` as an argument that is thread safe and allows for shared ownership and interior mutability.

2.4 Client::reauth API Design

The `reauth` is an API that enables users to send Auth packet for re-authentication purposes. The API is defined as follows:

```
pub async fn reauth(&self, properties: Option<AuthProperties>) ->
Result<(), ClientError>;
```

2.5 Example Usage

To use the enhanced authentication feature with a need to exchange Auth packet between Client and Server, users have to define a struct that implements the **`AuthManager`** trait and set it through the *`set_auth_manager`* API. The struct should contain the data structure and logic that are needed for calculating the client messages for the enhanced authentication method. The struct should also implement the *`auth_continue`* method that takes the server authentication data and returns the client authentication data according to the authentication protocol.

Here is an example code that shows how to use the enhanced authentication feature. The code defines a struct called **`ExampleAuthManager`** that implements the **`AuthManager`** trait and set it as the authentication manager through *`set_auth_manager`* API.

```
struct ExampleAuthManager{
    // Define the data structure that needed for calculating client messages for
enhanced authentication.
}

impl ExampleAuthManager {
    fn new(user: &str, password: &str) -> ExampleAuthManager{
        ExampleAuthManager{
            // Initiate the application authentication manager.
        }
    }

    fn auth_start(&mut self) -> Result<Option<Bytes>, Error>{
        // Calculate the client first message and return the data.
    }
}

impl AuthManager for ExampleAuthManager {
    fn auth_continue(&mut self, auth_prop: Option<AuthProperties>) ->
Result<Option<AuthProperties>, String> {
        // Calculate client message according to the received server data and return
the data.
    }
}

void connection()
{
    ...
    let mut authmanager = AuthManager::new("username", "password");
    let client_first = authmanager.auth_start().unwrap();

    let mut mqttoptions = MqttOptions::new("id", "hostname", "port");
    mqttoptions.set_authentication_method("Authentication Method".to_string());
    mqttoptions.set_authentication_data(client_first);
    mqttoptions.set_auth_manager(Arc::new(Mutex::new(authmanager)));
    ...
}
```

- Example code for authentication using **SCRAM-SHA-256** method.

```
use rumqttc::v5::mqttbytes::{QoS, v5::AuthProperties};
use rumqttc::v5::{AsyncClient, MqttOptions, AuthManager};
use tokio::task;
use std::error::Error;
use std::sync::{Arc, Mutex};
use bytes::Bytes;
use scram::ScramClient;
use scram::client::ServerFirst;
use flume::bounded;
```

```
#[derive(Debug)]
struct ScramAuthManager <'a>{
    user: &'a str,
    password: &'a str,
    scram: Option<ServerFirst<'a>>,
}

impl <'a> ScramAuthManager <'a>{
    fn new(user: &'a str, password: &'a str) -> ScramAuthManager <'a>{
        ScramAuthManager{
            user,
            password,
            scram: None,
        }
    }

    fn auth_start(&mut self) -> Result<Option<Bytes>, String>{
        let scram = ScramClient::new(self.user, self.password, None);
        let (scram, client_first) = scram.client_first();
        self.scram = Some(scram);

        Ok(Some(client_first.into()))
    }
}

impl <'a> AuthManager for ScramAuthManager<'a> {
    fn auth_continue(&mut self, auth_prop: Option<AuthProperties>) -> Result<
Option<AuthProperties>, String> {
        // Unwrap the properties.
        let prop = auth_prop.unwrap();

        // Check if the authentication method is SCRAM-SHA-256
        if let Some(auth_method) = &prop.method {
            if auth_method != "SCRAM-SHA-256" {
                return Err("Invalid authentication method".to_string());
            }
        } else {
            return Err("Invalid authentication method".to_string());
        }

        if self.scram.is_none() {
            return Err("Invalid state".to_string());
        }

        let scram = self.scram.take().unwrap();

        let auth_data = String::from_utf8(prop.data.unwrap().to_vec()).unwrap();
```

```rust
        // Process the server first message and reassign the SCRAM state.
        let scram = match scram.handle_server_first(&auth_data) {
            Ok(scram) => scram,
            Err(e) => return Err(e.to_string()),
        };

        // Get the client final message and reassign the SCRAM state.
        let (_, client_final) = scram.client_final();

        Ok(Some(AuthProperties{
            method: Some("SCRAM-SHA-256".to_string()),
            data: Some(client_final.into()),
            reason: None,
            user_properties: Vec::new(),
        }))
    }
}

#[tokio::main(flavor = "current_thread")]
async fn main() -> Result<(), Box<dyn Error>> {

    let mut authmanager = ScramAuthManager::new("user", "password");
    let client_first = authmanager.auth_start().unwrap();
    let authmanager = Arc::new(Mutex::new(authmanager));

    let mut mqttoptions = MqttOptions::new("client_id", "hostname", port);
    mqttoptions.set_authentication_method(Some("SCRAM-SHA-256".to_string()));
    mqttoptions.set_authentication_data(client_first);
    mqttoptions.set_auth_manager(authmanager.clone());
    let (client, mut eventloop) = AsyncClient::new(mqttoptions, 10);

    let (tx, rx) = bounded(1);

    task::spawn(async move {
        client.subscribe("rumqtt_auth/topic", QoS::AtLeastOnce).await.unwrap();
        client.publish("rumqtt_auth/topic", QoS::AtLeastOnce, false, "hello
world").await.unwrap();

        // Wait for the connection to be established.
        rx.recv_async().await.unwrap();

        let client_first = authmanager.clone().lock().unwrap().auth_start().unwrap();
        let properties = AuthProperties{
            method: Some("SCRAM-SHA-256".to_string()),
            data: client_first,
            reason: None,
            user_properties: Vec::new(),
        };
        client.reauth(Some(properties)).await.unwrap();
    });

    loop {
        let notification = eventloop.poll().await;

        match notification {
            Ok(event) => {
                println!("{:?}", event);
                match(event){
                    rumqttc::v5::Event::Incoming(rumqttc::v5::Incoming::ConnAck(_))
=> {
                        tx.send_async("Connected").await.unwrap();
                    }
                    _ => {},
                }
            }
            Err(e) => {
                println!("Error = {:?}", e);
                break;
            }
        }
    }

    Ok(())
}
```

- Example Code for Microsoft Entra **JWT** authentication.

```
use rumqttc::v5::mqttbytes::v5::AuthProperties;
use rumqttc::v5::{AsyncClient, MqttOptions, mqttbytes::QoS};
use rumqttc::{TlsConfiguration, Transport};
use tokio::task;
use tokio_rustls::rustls::ClientConfig;
use std::error::Error;
use std::sync::Arc;

#[tokio::main(flavor = "current_thread")]
async fn main() -> Result<(), Box<dyn Error>> {
    let pubsub_access_token = "";

    let mut mqttoptions = MqttOptions::new("client_id", "hostname", port);
    mqttoptions.set_authentication_method(Some("OAUTH2-JWT".to_string()));
    mqttoptions.set_authentication_data(Some(pubsub_access_token.into()));

    // Use rustls-native-certs to load root certificates from the operating system.
    let mut root_cert_store = tokio_rustls::rustls::RootCertStore::empty();
    root_cert_store.add_parsable_certificates(
        rustls_native_certs::load_native_certs().expect("could not load platform
certs"),
    );

    let client_config = ClientConfig::builder()
        .with_root_certificates(root_cert_store)
        .with_no_client_auth();

    let transport =
Transport::Tls(TlsConfiguration::Rustls(Arc::new(client_config.into())));

    mqttoptions.set_transport(transport);

    let (client, mut eventloop) = AsyncClient::new(mqttoptions, 10);

    task::spawn(async move {
        client.subscribe("topic1", QoS::AtLeastOnce).await.unwrap();
        client.publish("topic1", QoS::AtLeastOnce, false, "hello
world").await.unwrap();

        // Re-authentication test.
        let props = AuthProperties{
            method:Some("OAUTH2-JWT".to_string()),
            data:Some(pubsub_access_token.into()),
            reason:None,
            user_properties:Vec::new(),
        };

        client.reauth(Some(props)).await.unwrap();
    });

    loop {
        let notification = eventloop.poll().await;

        match notification {
            Ok(event) => println!("{:?}", event),
            Err(e) => {
                println!("Error = {:?}", e);
                break;
            }
            }
        }

        Ok(())
    }
```