Proton Authentication and Encryption (SASL and SSL support)

The Qpid Proton C transport code is responsible for receiving and sending all the protocol bytes into and out of the Proton-C engine. It encapsulates all the protocol processing. Effectively turning the wire level protocol into more abstract AMQP model state changes and vice versa turning AMQP model state changes into wire level protocol to send.

The transport code is divided into 3 layers that divide up the protocol processing:

- SSL
- SASL
- AMQP

Loosely speaking, SSL is mostly associated with encrypting the connection, SASL with authenticating the connection, and of course AMQP carries the actual messaging protocol we're "really" interested in. However despite common knowledge, actually SSL and SASL can both do encryption and authentication of the connection. It's important to note that the SSL and SASL layers are optional in the sense that if authentication and/or encryption are not required then they need not be used - This will be covered in more detail later on.

In the current Proton codebase (0.8) the SSL layer is fairly comprehensively implemented (using the OpenSSL and Windows SChannel implementations) and allows access to both encryption and authentication (although SSL authentication is somewhat complex to set up and use).

Conversely the SASL layer is very rudimentary and essentially punts to the application to handle the SASL protocol interchange. The SASL code does have some built in capability to use the ANONYMOUS mechanism and some code to simplify implementing the PLAIN SASL protocol exchange. Using the Proton-C 0.9 SASL codebase requires the application to directly read the SASL protocol bytes from the peer process them and send raw protocol bytes back to the peer.

I have been re-implementing the SASL layer using the Cyrus-SASL library which supports many SASL mechanisms via a range of plugins (including ANONYMOUS, EXTERNAL, PLAIN, DIGEST-MD5, CRAM-MD5, GSSAPI etc.) and which is extensible to new SASL mechanisms by writing new mechanism plugins.

This work has led me re-evaluate some of the transport API as a whole as well as more specifically the SASL API which more directly needs reworking because of these changes.

Specifically I'd like to somewhat simplify using authentication and encryption with the transport by unifying the most commonly used concepts into the transport code and only requiring delving directly into the SSL/SASL APIs for more complex uses.

API Changes

From the authentication point of view the API flow is much simpler than before:

Instead of the Proton-C 0.9 situation where the application has to directly read and write AMQP SASL frames using `pn_sasl_recv()`, `pn_sasl_send()` etc. the meat of the protocol interchange happens behind the scenes; the application only has to set up authentication parameters, and then allow Proton-C to handle SASL without application intervention.

At the client the parameters that can be set up include username/password (with `pn_connection_set_user/pn_connection_set_password()`), if necessary the SASL mechanisms used can be set (`pn_sasl_allowed_mechs()`).

By default the server will adapt to whatever layers the client uses to communicate. But the server application can specify that the connection must be encrypted or authenticated (using `pn_transport_require_encryption()` and `pn_transport_require_auth()`). Also if required the server application can force the SASL layer to use the ANONYMOUS mechanism or to exclude some mechanisms that might be installed on the system (by using `pn_sasl_force_anonymous()` or `pn_sasl_exclude_mechs()`). The location and name of the configuration file used by the SASL implementation can be changed by using the `pn_sasl_config_name()` and `pn_sasl_config_path()` APIs.

The outcome of the authentication process at both server and client sides is indicated by a transport event. In the case of authentication failure it will be a `PN_TRANSPORT_ERROR` event, and the case of success it will a PN_TRANSPORT_AUTHENTICATED event. Once the authentication event has been received the server can discover who it is talking to by using the `pn_transport_get_user()` API. If required the mechanism used can be recovered using `pn_sasl_get_mech()`.

Note that on the server you can also use the PN_CONNECTION_REMOTE_OPEN event to signal that authentication has succeeded if you don't need to deal with authentication by itself.

SASL

The biggest functional changes happen in the SASL layer code where the API has largely changed, very little backwards compatibility has been kept, because it is fairly clear that very few (if any) people have been using the current API to implement their own SASL mechanisms.

If this turns out to be untrue, then we could add some further measure of backwards compatibility as required.

Functions Removed

- void pn_sasl_client(pn_sasl_t *sasl)
- void pn_sasl_server(pn_sasl_t *sasl)

These functions were used to specify the SASL layer as either a client or server, they have been deprecated since functionality went into the transport code to specify whether it is an authentication client or server. Currently, on creation, the SASL layer will determine from the transport whether it should be a server or a client. As there is a good deal of SASL churn in these current changes this is a good opportunity to remove these deprecated APIs.

- `pn_sasl_state_t pn_sasl_state(pn_sasl_t *sasl)`
- `size_t pn_sasl_pending(pn_sasl_t *sasl)`
- `ssize_t pn_sasl_recv(pn_sasl_t *sasl, char *bytes, size_t size)`
- `ssize_t pn_sasl_send(pn_sasl_t *sasl, const char *bytes, size_t size)`

These functions are the ones which support the client directly reading and writing the SASL protocol frames. They are not needed as this functionality has been removed and superseded by internal functionality.

- `void pn_sasl_mechanisms(pn_sasl_t *sasl, const char *mechanisms)`
- `const char *pn_sasl_remote_mechanisms(pn_sasl_t *sasl)`

These functionality here has been replaced by `pn_sasl_exclude_mechs()` and `pn_sasl_get_mech()`.

- `void pn_sasl_plain(pn_sasl_t *sasl, const char *username, const char *password)`

This function is a helper function that constructs and sends a SASL PLAIN mechanism frame. As directly sending the frames is no longer the responsibility of the application it is not needed, however it is broadly replaced with `pn_transport_set_user_password()` which allows the client to set the authentication username and password.

- `void pn_sasl_allow_skip(pn_sasl_t *sasl, bool allow)`

This function is replaced with `pn_transport_require_auth(pn_transport_t *transport, bool required)` although the sense is reversed. By default anonymous connections are allowed. This is not really a change as previously there was no API guard against the SASL ANONYMOUS mechanism which is no better authenticated than not using SASL at all.

Functions Added

- `const char *pn_sasl_get_user(pn_sasl_t *sasl)`
  This is usually used at the the server end to find the name of the authenticated user. On the client it will merely return whatever user was passed in to `pn_transport_set_user_password()`.
  The returned value is only reliable after the `PN_TRANSPORT_AUTHENTICATED` event has been received. If the SASL layer was not negotiated then 0 is returned If the ANONYMOUS mechanism is used then the user will be "anonymous" Otherwise a string containing the authenticated user is returned.
- `const char *pn_sasl_get_mech(pn_sasl_t *sasl)`
  Return the selected SASL mechanism: The returned value is only reliable after the `PN_TRANSPORT_AUTHENTICATED` event has been received.
- `void pn_sasl_allowed_mechs(pn_sasl_t *sasl, const char *mechs)`
  Specify SASL mechanisms that are to be considered for authentication. This can be used on either the client or the server to restrict the SASL mechanisms that may be used.
  Multiple allowed mechanisms are space separated in the string that is passed into this function.
  As a special case on the client if the only mechanism allowed is "ANONYMOUS" then this will activate a mode ("Pipelined mode") where the client will assume that the server will accept anonymous SASL connections and will not wait for the SASL protocol interchange, but will just send all the relevant SASL frames unconditionally.
  However an important note is that if this is set on a client and the connected server really does require authentication then the server will fail and disconnect the connection with no useful feedback to the application as the SASL layer state will already be assuming it has authenticated.
  There is no similar server pipelined mode as it is not possible together with detecting and adapting to whatever AMQP protocol layers the connecting client is using.
- `void pn_sasl_config_name(pn_sasl_t *sasl, const char *name)`
  This is used to construct the SASL configuration filename. In the current implementation ".conf" is added to the name and the file is looked for in the configuration directory.
  If not set it will default to "proton-server" for a sasl server and "proton-client" for a client.
- `void pn_sasl_config_path(pn_sasl_t *sasl, const char *path)`
  This is used to tell SASL where to look for the configuration file. In the current implementation it can be a colon separated list of directories.
  The environment variable PN_SASL_CONFIG_PATH can also be used to set this path, but if both methods are used then this pn_sasl_config_path() will take precedence.
  If not set the underlying implementation default will be used.

Transport

Functions Added

- `const char *pn_transport_get_user(pn_transport_t *transport)`

  This is usually used at the the server end to find the name of the authenticated user, On the client it will merely return whatever user was passed in to `pn_connection_set_user()`.

  The returned value is only reliable after the PN_TRANSPORT_AUTHENTICATED event has been received.

  If a the user is anonymous (either no SASL layer is negotiated or the SASL ANONYMOUS mechanism is used) then the user will be "anonymous", otherwise a string containing the user is returned.
- `void pn_transport_require_auth(pn_transport_t *transport, bool required)`

  Set whether a non authenticated transport connection is allowed.

There are several ways within the AMQP protocol suite to get unauthenticated connections:

  - Use no SASL layer (with either no SSL or SSL without client certificates)
  - Use an SASL layer but the ANONYMOUS mechanism

The default if this option is not set is to allow unauthenticated connections.

- `bool pn_transport_is_authenticated(pn_transport_t *transport)`
  Returns whether the transport is authenticated.
  This property may not be stable until after the PN_CONNECTION_REMOTE_OPEN event is received.

- `void pn_transport_require_encryption(pn_transport_t *transport, bool required)`
  Set whether a non encrypted transport connection is allowed.
  There are several ways within the AMQP protocol suite to get encrypted connections:
  - Use SSL
  - Use SASL with a mechanism that supports security layers
  The default if this option is not set is to allow unencrypted connections.
- `bool pn_transport_is_encrypted(pn_transport_t *transport)`
  Returns whether the transport is encrypted.
  This property may not be stable until after the PN_CONNECTION_REMOTE_OPEN event is received.

Connection

Functions Added

- `void pn_connection_set_user(pn_connection_t *connection, const char *user)`

  Set the authentication username for a client transport

  If not set then no authentication will be negotiated unless the client sasl layer is explicitly created (this would be for something like Kerberos where the credentials are implicit in the environment, or to explicitly use the ANONYMOUS SASL mechanism).
- `void pn_connection_set_password(pn_connection_t *connection, const char* password)`

  Set the authentication password for a client transport. Note that there is no way to retrieve the password from the API as them implementation is supposed to use it and then zero it out of memory as soon as it is finished with it.
- `const char *pn_connection_get_user(pn_connection_t *connection)`
  Get the authentication username for a client transport.

New Events

- PN_TRANSPORT_AUTHENTICATED
  This event is sent when a server successfully authenticates a client connection (or when it accepts an unauthenticated connection that does not need to be authenticated).On the client it is sent when the client successfully authenticates to the server. If encryption is configured for the connection, it will be also be established by the point at which this event is received. If the authentication or encryption handshake is unsuccessful then both server and client applications will receive a `PN_TRANSPORT_ERROR` event.

Authentication and Encryption Combinations

This table summarises the possible combinations of SSL and SASL use. The implicit assumption of the table is that there is no point in doing double encryption or double authentication so this is not captured in the table. Double encryption would be using both SSL and SASL to encrypt the connection and likewise double authentication would be using both for authentication.

Classically in protocols where SASL was not optional the way to avoid double authentication was to use the EXTERNAL SASL mechanism. With AMQP, SASL is optional, so if SSL is used for client authentication the SASL layer could be entirely omitted and so using EXTERNAL is not necessary.

Similarly in protocols where the SASL layer is not optional the ANONYMOUS mechanism is used when client authentication is not required. With AMQP as the SASL layer can just be entirely omitted, and the SSL layer if present used only for encryption (and server authentication).

Auth\Encryp	None	SSL	SASL
None	1.	Privacy only	
SSL		2.	
SASL	Auth only	4.	3.

In this table "auth" means client authentication.

- Entries in pink aren't allowed/don't make any sense
- All other entries have some use

The numbered boxes are the combinations that are reasonably important:

1. Anonymous, "public" connection
2. Encrypted connection with client certificate
3. Kerberos or DIGEST-MD5
4. Usually SSL encryption with PLAIN authentication.

- All of the SSL encryption possibilities have some use and all could be used to authenticate the server for the client to avoid "man-in-the-middle" attacks.
- The right 2 columns correspond to an *encrypted* connection
- The bottom 2 rows correspond to an *authenticated* connection

Current State of Code

A substantial portion of this work is complete and ready to be included for the 0.10 Proton release (subject to code review of course). You can follow along with my changes by looking in my Github Proton repository:

https://github.com/astitcher/qpid-proton/commits/PROTON-334

The items of work that will remain to be completed are:

- ☐ Tie the SSL code into the more unified transport API:
  The current code base has no changes in the SSL code for these changes.
- ☐ Consequently, `pn_transport_require_encryption()` is not implemented.
- ☐ There needs to be a way to inject a byte stream that already comes from SSL without autodetection
  as this is how the qpidd broker will be able to use SSL with this SASL implementation.
- ☐ Using SASL for encryption is currently not implemented.
- ☐ Example code.
- ☐ An implementation of the PLAIN SASL mechanism that can be used when Cyrus SASL is not available.
  This will involve some application interaction as the application will have to authenticate the user, password combination itself and signal to proton whether the incoming connection is authenticated or not.
  The likely mechanism will be to send a transport event and for the application to use pn_sasl_done() to signal the outcome.