SAS® Event Stream Processing 4.2: Security

Encryption on Sockets

Overview to Enabling Encryption

You can enable encryption on TCP/IP connections within an event stream processing engine. Specifically, you can encrypt the following:

- connections that are created by a client using the C, Java, or Python publish/subscribe API to connect to an event stream processing server.
- connections that are created by an adapter connecting to an event stream processing server.
- connections that are created by a file and socket connector or adapter that acts as a socket client or server. In this case, the TCP peer can be another file and socket connector or adapter, or a third-party socket application.

  Note: Suppose a file and socket connector or adapter connects to a SAS LASR Analytic Server. Suppose further that the server runs on a Hadoop Distributed File System (HDFS) name node in order to access SAS HDAT files. In this case, encryption is not supported.

You must meet the following requirements in order to enable encryption:

- The OpenSSL libraries must be installed on all computer systems that run the client and server. When you install SAS Event Stream Processing Encryption and Authentication Overlay, you install OpenSSL.
- The `DFESP_SSLPATH` environment variable must be defined with the path to the OpenSSL shared object or DLL.

  Note: For a Java publish/subscribe client or adapter, the value of `DFESP_SSLPATH` is not important. SSL is supported natively in Java. Nevertheless, `DFESP_SSLPATH` must be defined.

- The proper SSL certificates must be installed on the client and server. If encryption is not enabled through DFESP_SSLPATH, the installation runs successfully without OpenSSL being installed.

Understanding SSL Certificate Requirements

Consider the following SSL certificate requirements in order to enable encryption:

- The required SSL certificates differ for client and server. You must copy the needed certificate files into **$DFESP_HOME/etc**.
- The server is an engine with publish/subscribe enabled or a file and socket connector or adapter running as a server. The server requires the following files in **$DFESP_HOME/etc**:
  - **server.pem**

    This file must contain a concatenation of a certificate and private key.
  - **key.passphrase** (only when the key is password-protected)
- A client is a C, Java, or Python publish/subscribe client, an adapter, or a file and socket connector or adapter running as a client. The client requires the **ca.pem** file in **$DFESP_HOME/etc**. This file must contain a certificate used to verify the received server side certificate.

  For certificates signed by a Certificate Authority (CA), the certificate of the signer must be present. For self-signed certificates, the certificate can be a copy of the certificate in **server.pem**.

Production traffic must use only certificates that are signed by a CA.

The event stream processing client and server forces the negotiated encryption protocol and cipher suite to be TLSv1.2 compliant.

Understanding the SSL Handshake Process

When DFESP_SSLPATH is defined but SSL certificates cannot be found or the OpenSSL library cannot be loaded, a fatal error is logged at start-up. If there are no start-up errors, the publish/subscribe server indicates whether SSL is enabled or disabled through an INFO level log message logged when the server starts up.

A client that is enabled for SSL logs a handshake-status INFO-level message when it initiates its connection. When a client or server is negotiating SSL and its peer is not, or when the SSL handshake itself fails, the connection fails and an error message is logged.

Enabling Authentication on Socket Connections

Overview

You can require authentication for TCP/IP clients that connect to an event stream processing engine. You must install the OpenSSL libraries on your event stream processing server in order to implement authentication. You can install the SAS Event Stream Processing Encryption and Authentication Overlay package to install OpenSSL.

Authentication applies to the following event stream processing engine APIs:

- the publish/subscribe API
- connections created by a client that uses the C, Java, or Python publish/subscribe API to connect to an event stream processing engine
- connections created by an adapter that connects to an event stream processing engine

- the XML server HTTPS API
- connections created by the XML client (`dfesp_xml_client`) to communicate with the XML server using the HTTPS protocol

To implement authentication, an event stream processing server must be enabled for publish/subscribe operations.

When enabled on a server, authentication is a global and permanent setting, so all clients that connect to that server must be authenticated. This applies to all client operations that are supported by the publish/subscribe API. It also applies to queries and all other client/server requests that establish a unique TCP connection. When authentication fails, a client is disconnected and an error message is logged on both the client and server.

Similarly, a client that requests authentication to a server that is not enabled for authentication results in client disconnection. There is a corresponding error message.

Token Method Authentication

The token method authentication mechanism requires a signed JSON Web Token obtained from an OAuth 2.0/ OpenID Connect compliant server. This token is supplied to the publish/subscribe API or adapter by the user. The token is then passed in compact serialization form (base64url encrypted) from client to server. It is parsed and validated by the server.

SAS Event Stream Processing requires that the token be obtained from a supported OAuth server. This is currently limited to the Cloud Foundry (CF) User Account and Authentication (UAA) Server. You must request a token from the CF UAA server, and then provide that token to the client.

For authentication certification, tokens are generated by invoking a REST request to a locally installed CF UAA server through curl. The REST request invokes the implicit grant with credentials flow. For more information, see the CF UAA documentation. For an example, see CF UAA Client/Server Information.

The following resources provide more information:

- JSON web tokens: https://self-issued.info/docs/draft-ietf-oauth-json-web-token.html
- OAuth 2.0: https://tools.ietf.org/html/rfc6749
- OpenID Connect: http://openid.net/specs/openid-connect-core-1_0.html
- CF UAA: https://github.com/cloudfoundry/uaa

Authentication Using a User Name and Password

This authentication mechanism requires simple user name and password credentials to be provided by the user. These credentials must be valid when passed by the event stream processing server to a SASLogon service.

The sequence of processing is as follows:

1. A user supplies credentials to the publish/subscribe API, adapter, or HTTP client.
2. Credentials are passed unmodified to the event stream processing server.
3. The server passes credentials unmodified in a REST request to the configured SASLogon service.
4. The server returns the result of the authentication request to the client.

SAS Event Stream Processing requires that the user at a minimum provide a user name in the publish/subscribe URL. Alternatively, the password can be provided directly in the publish/subscribe URL. Otherwise, the event stream processing client API searches an `.authinfo` or `.netrc` file in the client's local filesystem for a password that matches the provided user name. Either way, the password can be clear text or SAS encoded.

Token Method Server Requirements

Authentication is enabled on an event stream processing server by passing a client ID string when initializing the engine. In the C++ modeling API, this is a parameter in the `dfESPengine::initialize()` call. To enable authentication, replace the current `pubsub_ENABLE(`*`portNum`*`)` parameter with `pubsub_ENABLE_OAUTH(`*`portNum`*`, `*`clientId`*`)`.

When running an XML server, enable authentication by including the **`-auth`** ***`clientId`*** command-line parameter.

This *clientId* must match the CF UAA client_id used when requesting a token from the CF UAA server. For more information, see Token Validation.

In addition, the server must contain the public key used to sign the token in its local file system in $DFESP_HOME/etc/oauth/pubkey.pem. For more information, see Token Validation.

Any error in token validation causes the server to return an error code to the client. The client then disconnects from the server.

Token Method Client Requirements

A client requests an authenticated connection by passing a token to the server. You can provide the token to the client in one of two ways:

- Pass it in the publish/subscribe or adapter URL through the following optional element:

  `?oauth_token`

  This element must follow the *`host`*`:`*`port`* part of the URL, as follows:

  `dfESP:/`*`host`*`:`*`port`*`?oauth_token=`*`token`*`.....` The remainder of the URL is the same.
- Specify the complete path and filename of a file on the local file system that contains the token. When using the publish/subscribe API, call the corresponding C, Java, or Python publish/subscribe API method. The C method is `C_dfESPpubsubSetTokenLocation()` , and the Java and Python method is `setTokenLocation()`. When running an adapter, use the corresponding optional adapter configuration switch.
- When running an XML client, pass the token using the **`-auth-token`** or **`-auth-token-url`** command-line parameter.

Using both methods simultaneously is not allowed. It generates a publish/subscribe API error. The client passes the token opaquely to the server and waits for token validation results. If successful, the connection is established and further client server operations proceed normally. If unsuccessful, the client disconnects from the server.

Token Validation

The server validates multiple items in a received token:

Validated Item	Description
Token Signature	The server uses the OpenSSL libraries and the public key in **`$DFESP_HOME/etc/oauth/pubkey.pem`** to verify the signature in a received token. This public key must be the same key as the public key in the public-private key pair that is configured on the CF UAA server. You must generate, secure, and manually copy this key to the server. For more information, see CF UAA Client/Server Information.

Validated Item	Description	
Claims	`aud`	The client ID configured on the server must match the `aud` claim contained in all tokens received by the server. The value of the `aud` claim in a token is determined by the CF UAA client ID included in a request to the CF UAA server to obtain that token. A CF UAA administrator must configure an event stream processing specific client ID. You must specify that same ID when you start an authentication-enabled server. You must also specify that ID when you request a token to be used by a client connecting to that server. Server-specific privileges can be enforced by requiring server-specific client IDs for connections to that server.
	`exp`	Token validation fails when the token is expired.
	`user_name`	The `user_name` claim must be present in the token, but the server does not validate its value.

Server Requirements for Authentication Using a User Name and Password

This method of authentication is enabled on an event stream processing server by passing a SASLogon services URL when initializing the engine. In the C++ modeling API, this is through a parameter in the `dfESPengine::initialize()` call.

To enable authentication, replace the current `pubsub_ENABLE(`*`portNum`*`)` parameter with `pubsub_ENABLE_SASLOGONOAUTH(`*`portNum, sasLogonURL`*`)`. When running an XML server, enable this method of authentication by including the **`-saslogon-url`** command–line parameter.

In order for the event stream processing server to load and use the required SAS encryption libraries, you must set environment variable `TKERSA2_LIB_PATH` to the location of your SASFoundation libraries. Any error in authenticating the user credentials causes the server to return an error code to the client. The client then disconnects from the server.

Client Requirements for Authentication Using a User Name and Password

A client requests an authenticated connection by passing a user name and password to the server. You can provide these to the client in one of three ways:

- Pass them in the publish/subscribe or adapter URL through the following optional elements: `?username` and `?password` These elements must follow the ***`host:port`*** part of the URL, as follows: **`dfESP:/`*`host:port`*`?username=`*`username`*`?password=`*`password`***. The remainder of the URL is the same.
- Pass the user name as described in the previous way, but have the API extract the matching password from a local **`.authinfo`** or **`.netrc`** file. In this case environment variable `AUTHINFO` or `NETRC` should be set to the full path and filename. If an environment variable is not set, the client API looks for the file in the user's home directory.
- When running an XML client, pass the user name using the **`-auth-user`** command-line parameter.

The password can be encoded, but it does not have to be. If required, encode your password using the PWENCODE procedure. Be sure to include the `{SAS00x}` prefix from the PWENCODE result in the password string passed to the event stream processing client or included in your **`.authinfo`** or **`.netrc`** file. The client passes the credentials opaquely to the server and waits for SASLogon authentication results. If successful, the

connection is established and further client server operations proceed normally. If unsuccessful, the client disconnects from the server.

CF UAA Client/Server Information

The CF UAA server is an open-source package available from GitHub. After you install it, the following additional administrative steps are highly recommended:

- Generate a new private key using OpenSSL (**`openssl genrsa -out privkey.pem 1024`**, for example), and then generate a public key based on that private key (**`openssl rsa -pubout -in privkey.pem -out pubkey.pem`**, for example). Keep these keys secure.
- Configure the CF UAA token verification-key with the public key, and the CF UAA token signing-key with the private key. Then copy the public key to all event stream processing servers that should authenticate clients using tokens that are generated by this CF UAA server.
- Configure one or more CF UAA client IDs restricted for use only by users running an event stream processing server or client
- Register CF UAA user name and password credentials for users requiring tokens for use by an event stream processing client.

Once configured, the steps required to obtain and use tokens for authenticated connections are as follows:

- To connect a client to a specific server, obtain a token from a CF UAA server configured with the same public key used by the event stream processing server. The token request must contain the same CF UAA client ID that you used to enable authentication on the server. You can choose the method of requesting a token from CF UAA.
- Extract the token from the response and provide it to your client as described in Token Method Client Requirements.
- You can reuse a single token indefinitely when connecting to the same server, as long as that token remains unexpired.

Here is an example of a REST request invoked through curl to obtain a token from a CF UAA server through an implicit grant with credentials:

```
curl -v -H "Accept: application/json" -H "Content-Type: application/x-www-form-
urlencoded" "http://myhost:8080/uaa/oauth/authorize?
client_id=myclientid&response_type=token&scope=openid&redirect_uri=http://
localhost/hello" -d "credentials=%7B%22username%22%3A%22myusername%22%2C%22password
%22%3A%22mypassword%22%7D"
```

When the REST response contains a successful “302 Found” response, you can find the token in the `&access_token` portion of the `Location` field of the REST response.