SAS® Event Stream Processing 5.2: Security

Overview

Configure encryption and authentication by setting parameters in the security configuration file, **security-properties.yml**. This file resides in the configuration directory.

Whenever you start an ESP server or client on the command line without specifying arguments that govern security, the server and client use values specified in this file. Any values passed through a command–line argument governing security override values specified in this file.

Configuration Directory

After you deploy SAS Event Stream Processing, configuration files are located in a specific location:

Table 1 *Default Location of Configuration Files*

Operating System	Location
Linux	**/opt/sas/viya/config/etc/SASEventStreamProcessingEngine/default**
Windows	**%ProgramData%\SAS\Viya\etc\SASEventStreamProcessingEngine\default**

On Windows, if the environment variable ProgramData is defined, it is placed at the beginning of the configuration directory's path, before **\SAS**.

Note: To edit configuration files, you must be a member of the `sas` group.

Enabling Encryption on TCP/IP Connections

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 that uses the C, Java, or Python publish/subscribe API to connect to an event stream processing server.
- connections that are created by an adapter that connects 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 it can be a third-party socket application.

  Note: When a file–and–socket connector or adapter connects to a SAS LASR Analytic Server and that server runs on a Hadoop Distributed File System (HDFS) NameNode, encryption is not supported.

To enable encryption, the OpenSSL libraries must be installed on all computer systems that run the ESP server and clients. Version 1.0.2 or higher of the Transport Layer Security (TLS) protocol is required to take advantage of ECDH support for encryption ciphers used in encrypted connections.

Note: All discussion of TLS is also applicable to the predecessor protocol, Secure Sockets Layer (SSL).

You must install the proper TLS or SSL certificates on the ESP server and client.

Understanding TLS Certificate Requirements

To enable encryption, set the following properties in the security configuration file on the ESP server and client:

```
security:
  pubsub_ssl_enabled: false 1
  http_ssl_enabled: false
  server:
    auth: null
    server_cert_file: server.pem 2
    passphrase: passphrase 3
  client:
    ca_cert_file: ca.pem 4
    trust_selfsigned: false 5
```

1 Set `pubsub_ssl_enabled` and `http_ssl_enabled` to **`true`** to turn on TLS. By default, these are set to **`false`**.

2 The file specified here must contain a concatenation of a certificate key and a private key. It must be located in the configuration directory. SAS Event Stream Processing ships a file named `server.pem` that is used by default.

3 You can specify a *passphrase* to protect the `server_cert_file`. You can leave this blank.

4 The file specified here contains the certification authority (CA) certificate used by the client to validate the server's certificate during the TLS handshake. It must be located in the configuration directory. SAS Event Stream Processing ships a file named `ca.pem`, which is used by default. When the file is not found, a client attempts to use the system's default CA certificates. These default certificates must be in the configuration directory.

5 To trust a self-signed certificate, set `trust_selfsigned` to **`true`**. By default, this is set to **`false`**.

Note: Do not set `trust_selfsigned` on Windows systems or Java clients. On Windows systems, import the CA certificate into the trusted CA certificate store. On Java clients, import the CA certificate into the trusted keystore.

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

Note: You can change the path of the location of the security configuration file through the DFESP_SEC_YAML environment variable.

Note: When you start an event stream processing server, it can serve as a client to other servers. When a client needs a specific CA certificate file to validate the server, a **`ca_cert_file`** is also required by the ESP server. When a CA certificate file is not specified, SAS Event Stream Processing attempts to use the system's installed certificates (for example, **`/etc/pki/tls/certs/ca-bundle.crt`**).

When the ESP server needs to communicate with a User Account and Authentication Service (UAA) server through RESTful API calls, the server acts as an HTTPS client to the UAA. The server expects that the UAA is signed by a CA. No specific CA certificate other than the one installed on the system is required to establish the connection.

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

Understanding the TLS Handshake Process

The ESP server and client implement one–way authentication. That is, the server does not require the client to send a client certificate. If you want to skip the server certificate verification, you can set `trust_selfsigned` to **`true`**.

If there are no start-up errors, the publish/subscribe server indicates whether TLS is enabled or disabled through an INFO–level log message logged when the server starts.

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

Enabling Encryption for SAS Event Stream Processing Studio

To enable the Transport Layer Security (TLS) protocol for SAS Event Stream Processing Studio and the clients that access it:

- enable TLS on the ESP server
- obtain the certificate authority (CA) file to import to your browser and to the Java keystore

By default, the CA file is `ca.pem`.

1. Obtain the CA file for the system where SAS Event Stream Processing Studio is installed and for the clients that access the user interface.
2. On the systems from which users access SAS Event Stream Processing Studio, import the client certificate to the certificates store of your preferred web browser.
3. On the system where SAS Event Stream Processing Studio is running, import the client certificate to the Java keystore by running the following command:

   ```
   $JAVA_HOME/jre/bin/keytool -importcert -keystore keystore-location -file path-to-file -storepass password -noprompt -alias alias
   ```

4

Here is an example that assumes that you use `ca.pem` as the CA file:

```
$JAVA_HOME/jre/bin/keytool -importcert -keystore /opt/sas/viya/config/etc/
SASSecurityCertificateFramework/cacerts/trustedcerts.jks -file ca.pem -storepass
changeit -noprompt -alias myalias
```

Note: Specify the command on a single line. Multiple lines are used here to improve readability.

4 Restart the SAS Event Stream Processing Studio service. Run the appropriate command:

For Red Hat Enterprise Linux 6.7:

```
sudo service sas-viya-espvm-default stop
sudo service sas-viya-espvm-default start
```

For Red Hat Enterprise Linux 7.*x* or SUSE Linux:

```
sudo systemctl stop sas-viya-espvm-default
sudo systemctl start sas-viya-espvm-default
```

Enabling Authentication on TCP/IP Connections

Overview

You can require authentication for TCP/IP clients that connect to an event stream processing engine. The OpenSSL libraries must be installed on your event stream processing server to implement authentication. To implement authentication, an event stream processing server must be enabled for publish/subscribe operations.

Authentication applies to the following event stream processing engine APIs:

- the publish/subscribe API
- the ESP server HTTPS and WSS API
- connections that are created by an adapter that connects to an event stream processing engine
- connections that are created by any client that uses the C, Java, or Python publish/subscribe API to connect to an event stream processing engine
- connections that are created by the ESP client (**`dfesp_xml_client`**) to communicate with the ESP server using the HTTPS and WSS protocol

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 the 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.

Set the type of authentication to enforce in the security configuration file:

```
security:
  #
  # Encryption/Authentication Configuration for clients and Server
  #
  pubsub_ssl_enabled: false
  http_ssl_enabled: false
  server:
    auth: null 1 # {null|oauth_local|saslogon|kerberos|oauth2}
```

```
    server_cert_file: server.pem
    passphrase: passphrase
  client:
    auth: null # 1{null|oauth_local|oauth2.password|oauth2.client_credentials|saslogon|kerberos}
    ca_cert_file: ca.pem
    trust_selfsigned: false
```

1 Set `auth` to one of five possible values: **`null`** (the default value, to disable authentication), **`oauth_local`**, **`saslogon`**, **`Kerberos`**, or **`oauth2`**. Specify **`oauth_local`** for a stand-alone ESP server and clients. This authentication method works with an existing token. Using **`oauth2`** requires the existence of a running UAA server.

Note: Only Oauth2 cloud authentication is supported on ARM-based devices.

You can override security settings for an ESP server when you start it with the **`-auth`** command–line argument. For more information, see *SAS Event Stream Processing: Using the ESP Server*.

You can use one of two command–line arguments to override authentication settings for the client:

- Specify the authentication type through the URL for publish/subscribe clients such as connectors and adapters. The URL specification requires `?`*`auth_type`* after `dfESP://`*`host`*`:`*`port`* , where *auth_type* can be one of the following values: **`oauth_token`**, **`username`**, **`kerberos_servicename`**, or **`grant_type`**.
- Specify **`-auth`** ***`mechanism`*** on the command line for **`dfesp_xml_client`**.

You can display the current authentication mechanism by entering the client command on the console without any arguments. If you do not set the authentication mechanism on the command line, the client searches for the configuration file and uses it to determine the authentication configuration.

OAuth Token Method Authentication

Overview

Note: OAuth token method authentication is not supported on ARM-based devices.

The OAuth 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.

For OAuth local authentication, SAS Event Stream Processing requires that the token be obtained from 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.

You can manually request a token 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 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

OAuth Token Method Server Requirements

Authentication is enabled on an event stream processing server by passing a client ID string or using the security property file 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)`.

To enable OAuth local authentication on the server, set the following properties in the security configuration file .

```
security:
   ...
   server:
      auth: oauth_local
      ...
   oauth_local:
      server:
         client_id:client_id 1
         key_file: pubkey.pem 2
```

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

2 For `key_file`, specify the public key used to sign the token in its local file system in **`/opt/sas/viya/config/etc/SASEventStreamProcessingEngine/default/pubkey.pem`** (Linux) or **`%ProgramData%\SAS\Viya\SASEventStreamProcessingEngine\default\pubkey.pem`** (Windows). For more information, see "OAuth 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.

OAuth Token Method Client Requirements

To enable OAuth local authentication on a client, set the following properties in the security configuration file

```
security:
   ...
   client:
      auth: oauth_local
   ...
   oauth_local:
      client:
         token: null
         token_file: token.txt
```

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

- Pass it in the publish/subscribe or adapter URL using 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.

- Use the **`-auth oauth-token`** or **`-auth oauth-token-url`** command-line parameter of the ESP client.

Using more than one method 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. When successful, the connection is established and further client server operations proceed normally. When unsuccessful, the client disconnects from the server.

OAuth 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 **`/opt/sas/viya/config/etc/SASEventStreamProcessingEngine/default/pubkey.pem`** (Linux) or **`%ProgramData %\SAS\Viya\SASEventStreamProcessingEngine\default\pubkey.pem`** (Windows) 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 Server Information".	
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.

CF UAA 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 "OAuth 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.

OAuth 2.0 Cloud Authentication and Authorization

OAuth 2.0 is an industry-standard protocol that supersedes the original OAuth protocol. OAuth 2.0 is supported by the User Account and Authorization (UAA) server and is widely used in cloud deployments to support complex authentication and authorization services. The information that follows assumes the existence of a running UAA server, which typically is provided as a service by the cloud. The ESP server and client must be able to communicate with the UAA server to validate and obtain the token.

The ESP server supports OAuth 2.0 grant types: resource owner password and client credentials. Typically, an ESP adapter or a REST client acts as a client to access a resource server (in this case, an ESP server). The resource server is deployed in the cloud and is protected by an authorization server (such as the server provided by Cloud Foundry). Authorization-related tasks are delegated to the UAA server. In this way, the resource server is less exposed to security-related information and therefore is less vulnerable to security risks.

To configure OAuth Cloud Authentication and Authorization, it is recommended to set values in the security configuration file. Alternatively, you can request a token manually and use the command line argument for OAuth local authentication.

The ESP Server requires the following properties in the security configuration file:

```
security:
    server:
        auth: oauth2
        ...
    oauth2:
        server:
        check_token_endpoint: URL 1
    ...
        client_id: id 2
        client_secret: secret
```

1 Contact the administrator of the UAA server to obtain the `check_token_endpoint`. When you specify a value of **`$uaa.credentials.uri`** here, the ESP server tries to parse the VCAP environment variable exposed to it by Cloud Foundry using the JSON format.

2 Register with the UAA server to obtain a valid `client_id` and `client_secret`.

When an OAuth token is received by the ESP server, the server uses its `client_id` and `client_secret` to forward the token to the UAA server. The UAA server issued this token through a REST call to check whether it was valid and authorized to access the server.

Typically, the UAA server enables TLS (that is, it uses the HTTPS protocol). If the UAA server certificate is signed by a certification authority (CA), then the system's installed CA certificates are able to validate it in most cases. If the ESP server is self-signed, you can manually obtain the certificate and specify its location through `ca_cert`.

You can also choose to use the command line **-auth** argument to override properties as follows:

dfesp_xml_server -auth "oauth2://?client_id=*id*&client_secret=*secret*"

You must have a valid OAuth token on each ESP client in order for it to communicate with the server. The token requires the following properties in the security configuration file:

```
security:
    ...
    client:
        auth: oauth2.password
    ...
    oauth2:
    ...
        client:
            server_url: URL
            grant_types:
                password:
                    username: name
                    password: password
                client_credentials:
                    client_id: id
                    client_secret: secret
```

ESP clients (for example, adapters and **dfesp_xml_client**) support password and client credentials grant types. To use the password grant type, you must have a valid user name, password, client ID, and client secret from the UAA. To use the client credentials grant type, you need only a valid client ID and secret.

You can submit arguments to override the property file when you start a client on the command line. If your client supports URL specifications, as C++ or Java adapters do, the URL is one of the following:

- **"dfESP://url?grant_type=client_credentials&client_id=*id*&client_secret=*secret*"**
- **dfESP://url?grant_type=password&username=*name*&password=*password*&client_id=*id*&client_secret=*secret*"**

The parameters after the grant type are optional. If you do not specify them, the ESP client obtains parameter values from the property file. If your client supports the **-auth** command line argument, you can specify one of the following:

- **-auth "oauth2://?grant_type==client_credentials&client_id=*id*&client_secret=*secret*"**
- **-auth "oauth2://?grant_type=password&username=*name*&password=*password*&client_id=*id*&client_secret=*secret*"**

As before, parameters after the grant type are optional.

Authentication Using a User Name and Password (SASLogon Service)

Overview

Note: This authentication mechanism is not supported on ARM-based devices.

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.

Server Requirements for Authentication Using SASLogon Services

This method of authentication is enabled on an ESP server by passing a SASLogon services URL when initializing the engine.

To set the authentication mechanism through the configuration file, specify the following properties in the security configuration file on the ESP server:

```
security:
   server:
      auth: saslogon 1
   ...
   saslogon:
      server:
         server_url: URL
         ca_cert: ca.pem
```

1 If you specify **$saslogon_server** for `server_url`, it uses the value of that environment variable.

When starting an ESP server on the command line, include the **-auth saslogon://*sasLogonURL*** command–line parameter.

SASLogon servers also support HTTPS.

By default, the ESP server requests a server certificate from the SASLogon server. If no valid certificate is provided, then the session terminates. When you set `security.trust_selfsigned=true`, the session proceeds normally even when no certificate is provided or when an invalid certificate is provided. This functionality is unavailable on Windows systems.

Client Requirements for Authentication Using SASLogon Services

A client requests an authenticated connection by passing a user name and password to the server. Specify the following properties in thesecurity configuration file on the client:

```
security:
   ...
   client:
      auth: saslogon
   ...
   saslogon:
   ...
      client:
         username: username
```

You can override these values when you start a client on the command line.

- Specify them on 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.

  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.
- 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 starting an ESP client, pass the user name using the **-auth saslogon://*username*** command-line parameter.

Kerberos Authentication

Overview

Note: Kerberos authentication is not supported on ARM-based devices.

Enabling Kerberos authentication requires that the ESP server and client reach a Kerberos server. That Kerberos server must support a service name to be used by the server and client.

When these conditions are met, a user can pass the same Kerberos service name to the ESP server and client. The appropriate Kerberos exchange is performed between client and server when the client connects. The client builds the full service principal name by combining the service name passed by the user with the host contained in the publish/subscribe URL passed by the user.

Server Requirements for Kerberos Authentication

Enabling Kerberos authentication on an event stream processing server requires passing a Kerberos service name when you initialize the engine. In the C++ modeling API, you initialize the engine through a parameter in the `dfESPengine::initialize()` call. To enable authentication, replace the current `pubsub_ENABLE(portNum)` parameter with `pubsub_ENABLE_KERBEROS(portNum, serviceName)`.

The Kerberos server environment must contain a readable KeyTab file, which is usually found in **/etc/krb5.keytab**.

Set the following properties in the security configuration file on the ESP server:

```
security
   server:
      auth: kerberos
   ...
   kerberos:
      service_name: serviceName
```

You can override these values with the **-auth kerberos://*serviceName*** command–line parameter.

Note: The event stream processing server does not support Kerberos authentication on the Microsoft Windows platform.

Client Requirements for Kerberos Authentication

A client requests an authenticated connection by passing a service name to the server.

Set the following properties in the security configuration file on the clients, include Java and C++ clients:

```
security:
   client:
      auth: kerberos
   kerberos:
      service_name: serviceName
```

You can override these values using the **-auth kerberos://*serviceName*** command-line parameter. The server verifies that it was configured with the same service name before initiating the Kerberos protocol exchange.

When you start a client on the command line, you can specify the service name through the **?kerberos_servicename** element. This element must follow the ***host*:*port*** part of the URL. For example:

dfESP://*host*:*port*?kerberos_servicename=*servicename*.

The remainder of the URL is the same.

You might have to run **kinit** before you run the client to refresh the local token cache.

If the client is a Java client, a jaas.conf file must be present in the SAS Event Stream Processing configuration directory. It should be a copy of the jaas.conf file that is already used by other Java clients on the client machine. The file must contain these lines:

```
com.sun.security.jgss.krb5.initiate {

com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true
   renewTGT=true
   doNotPrompt=true;
};
```

If a Java client fails authentication, you can use these debug properties on the Java command line:

–Dsun.security.krb5.debug=true -Dsun.security.jgss.debug=true

If debugging results show the message "unsupported key type found the default TGT: 18", you might have to install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files.

Security Configuration File

The following sample file displays default values for security parameters:

```
security:
  #
  # Encryption/Authentication Configuration for clients and Server
  #
  pubsub_ssl_enabled: false 1
  http_ssl_enabled: false 2
  server:
    auth: null # {null|oauth_local|saslogon|kerberos|oauth2}
    server_cert_file: server.pem 3
    passphrase: passphrase
  client:
```

```
  auth: null # {null|oauth_local|oauth2.password|oauth2.client_credentials|saslogon|kerberos}
  ca_cert_file: ca.pem 4
  trust_selfsigned: false 5
#
# Authentication configuration parameters
#

# 1. Basic OAUTH2 authentication
#
oauth_local:
  server:
    client_id: clientid 6
    key_file: pubkey.pem
  client:
    token: null 7
    token_file: token.txt
    token_url: URL

# 2. Username/Password authentication via the SASlogon service
#
saslogon:
  server:
    server_url: URL
    #ca_cert: 8
        client:
    username: username

# 3. Kerberos authentication
#
kerberos:
  service_name: serviceName

# 4. OAuth2 (Cloud Foundry variant)
#
oauth2:
  server:
    check_token_endpoint: URL 9
  client:
    server_url: URL 10
    grant_types: 11
      password:
        username: name
        password: password
        client_credentials: null
        client_id: id
        client_secret: secret 12
```

1 Publish/subscribe clients and servers read this to enable Transport Layer Security (TLS). They use the same certifications as HTTP clients and servers.

2 HTTP clients and servers read this file to enable TLS.

3 This file is named `server.pem` by default. It should be located in the configuration directory and contain both certificate and private keys.

4 This file is named `ca.pem` by default. It should be located in the configuration directory. If it is not present, the system checks other installed certificates in the configuration directory.

5 This must be set to **true** here and in the SASLogon section of the security configuration file er to support SASLogon.

6 This is the client ID specified in the key file. The key file is named `pubkey.pem` by default. Obtain the ID from the User Account and Authentication Service (UAA) to verify that the token signature is signed.

7 The client first checks to see whether the token itself is specified here. If not, the client checks for the existence of the specified token file in the configuration directory. If that file does not exist. the client accesses the specified URL.

8 This specifies the location of the CA certificate.

9 This supports an environment variable and Cloud Foundry (for example, **$uaa.credentials.uri**).

10 Java adapters must import the certificate to a trusted keystore. The keystore must trust self-signed.

11 This supports resource owner password and client credentials.

12 This works with the UAA administrator to determine your *secret*.

Using Access Control

You can set up the event stream processing server to use explicit read/write permissions on engine, project, query, and window objects based on the user. You define these permissions in the file permissions.yml, which is located in your configuration directory . This file defines all permissions on all objects for all users for the server installation defined by **$DFESP_HOME**. The default installation contains a permissions.yml.sample file in your configuration directory.

Access control applies only to networked event stream processing clients, such as the ESP client, publish/subscribe clients, adapters, and Streamviewer. The user who runs the server is not limited in any way.

Note: You cannot use access control on the 64–bit ARM platform.

To enable access control, the server administrator can modify permissions.yml.sample to reflect the objects in the model and the users who need access. Then, the administrator can rename the sample file to permissions.yml.

If permissions.yml does not exist or cannot be read, then the event stream processing server runs without access control. At start-up, the event stream processing server sends an informational message that logs whether access control defined in permissions.yml is in effect.

If an event stream processing client does not provide a user name, then it is assumed to use “anonymous”. To provide a user name when connecting, an event stream processing client must authenticate against an authentication-enabled event stream processing server. Furthermore, the server must be configured for SASLogon style authentication. That is, a C++ server must use `pubsub_ENABLE_SASLOGONOAUTH(portNum, sasLogonURL)`, and an ESP server must specify **–auth saslogon** on the command line.

When a client is denied access, the server informs the client, closes the connection, and logs a warning message. The client logs an error message with the error code returned from the server.

A user whose name is not listed in permissions.yml or who attempts to access an object not listed in permissions.yml is denied access. The only supported permissions are Read and Write, and every user-object combination must have its Read and Write access explicitly defined as true or false.

In addition to users, permissions.yml can also contain groups, where group permissions are defined using the same syntax for users. Then, any user can be added to a group. A user can have explicitly defined permissions or can belong to a group but not both.

You can specify "*" (including the quotation marks) instead of the name of an engine, project, query, or window within permissions.yml. In this case, the complete set of corresponding engines, projects, queries, or windows at that level, and any objects contained within, assumes the read/write permissions that you specify for "*".

When an object is dynamically added to a running server (when an XML client loads a project, for example), the server grants Read and Write privileges to the user who created the object. The permissions.yml file is not modified.