



SAS[®] Event Stream Processing: エッジ環境でのセキュリティの 実装

2023.03 - 2024.02

このドキュメントは、ソフトウェアの追加バージョンに適用される場合があります。このドキュメントを [SAS Help Center](#) で開き、バナーのバージョンをクリックすると、使用できるすべてのバージョンが表示されます。

概要	2
TCP/IP 接続での暗号化の有効化	2
暗号化の有効化に関する概要	2
TLS 証明書要件について	3
TLS ハンドシェイクプロセスについて	4
TCP/IP 接続での認証の有効化	5
概要	5
OAuth トークン方式の認証	6
OAuth 2.0 クラウド認証および承認	9
ユーザー名とパスワードを使用した認証(SASLogon サービス)	11
Kerberos 認証	13
暗号化と認証の構成	15

概要

エッジシステムに直接 SAS Event Stream Processing for Edge Computing を配置する場合、エッジシステムで構成ファイルのセキュリティセクション、`esp-properties.yml` にパラメーターを設定して暗号化と認証を設定することができます。このファイルは構成ディレクトリにあります。

TCP/IP 接続での暗号化の有効化

暗号化の有効化に関する概要

イベントストリーム処理エンジン内の TCP/IP 接続で暗号化を使用可能にすることができます。具体的には、次のものを暗号化できます。

- C、Java、または Python パブリッシュ/サブスクライブ API を使用してイベントストリーム処理サーバーに接続するクライアントによって作成される接続。
- イベントストリーム処理サーバーに接続するアダプターによって作成される接続。
- ソケットクライアントまたはサーバーとして動作するファイルとソケットのコネクタまたはアダプタによって作成される接続。この場合、TCP ピアは別のファイルとソケットのコネクタまたはアダプタでも、サードパーティのソケットアプリケーションでも構いません。

注: ファイルとソケットのコネクタまたはアダプタが SAS LASR Analytic Server に接続し、そのサーバーが Hadoop 分散ファイルシステム(HDFS)の NameNode で動作する場合、暗号化はサポートされません。

暗号化を有効にするには、OpenSSL ライブラリが、ESP サーバーおよびクライアントを実行するすべてのコンピュータシステムにインストールされている必要があります。暗号化された接続で使用される暗号化暗号に対する ECDH サポートを利用するには、TLS (Transport Layer Security) プロトコルのバージョン 1.0.2 以降が必要です。

適切な TLS 証明書を ESP サーバーおよびクライアントにインストールする必要があります。

注: TLS に関するすべての議論は、先行プロトコルである SSL (Secure Sockets Layer) にも適用されます。

注: ECDH または ECDHE 暗号スイートを使用して ESP サーバへの TLS 接続を有効にするには、OpenSSL 1.0.2 が必要です。

TLS 証明書要件について

暗号化を有効にするには、ESP サーバーおよびクライアントにある次のセキュリティ構成プロパティを設定します。

```
security:
  pubsub_ssl_enabled: false 1
  http_ssl_enabled: false
  2_way_authentication: false
  trust_selfsigned: true 2
server:
  auth: null
  server_cert_file: server.pem 3
  ca_cert_file: # specify when 2_way_authentication enabled
  passphrase: passphrase 4
client:
  auth: null # {null|oauth_local|oauth2.password|oauth2.client_credentials|saslogon|kerberos}
  ca_cert_file: ca.pem 5
  client_cert_file: # specify when 2-way authentication is enabled
  passphrase:
```

- 1 TLS をオンにするには pubsub_ssl_enabled および http_ssl_enabled を **true** に設定します。デフォルトでは、これらは **false** に設定されています。
- 2 自己署名証明書を信頼するには、trust_selfsigned を **true** に設定します。

注: Windows システムまたは Java クライアントでは trust_selfsigned は設定しません。Windows システムでは、信頼された CA 証明書ストアに CA 証明書をインポートします。Java クライアントでは、信頼されたキーストアに CA 証明書をインポートします。

- 3 ここに指定されたファイルには、証明書鍵と秘密鍵の連結が含まれている必要があります。これは構成ディレクトリに置く必要があります。デフォルトでは、ファイル名は server.pem です。
- 4 server_cert_file を保護するために passphrase を指定することができます。これは空白のままにすることができます。
- 5 ここで指定されたファイルには、クライアントが TLS ハンドシェイク中にサーバーの証明書を検証するために使用する証明機関(CA)証明書が含まれています。これは構成ディレクトリに置く必要があります。デフォルトでは、ファイル名は ca.pem です。

ca.pem が見つからない場合、クライアントはシステムのデフォルト CA 証明書の使用を試みます。それらの証明書は、構成ディレクトリに存在する必要があります。

本番トラフィックは、CA によって署名された証明書のみを使用する必要があります。

注: イベントストリーム処理サーバーを起動すると、他のサーバーへのクライアントとして機能させることができます。クライアントがサーバーを検証するために特定の CA 証明書ファイルを必要とする場合、ESP サーバーでは **ca_cert_file** も必要になります。CA 証明書ファイルが指定されていない場合、SAS Event Stream Processing はシステムのインストール済み証明書を使用しようとします(例: /etc/pki/tls/certs/ca-bundle.crt)。

ESP サーバーが RESTful API 呼び出しを通じて User Account and Authentication Service (UAA)サーバーと通信する必要がある場合、サーバーは UAA の HTTPS クライアントとして機能します。サーバ

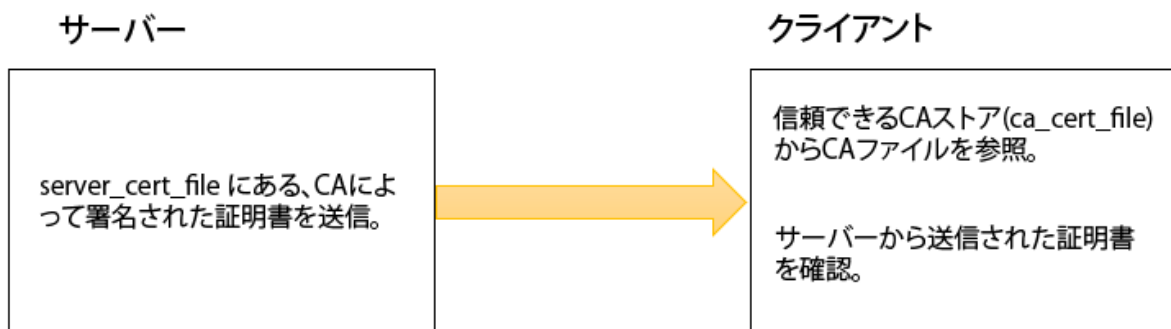
ーでは、UAA が CA によって署名されていると予期されます。接続を確立するには、システムにインストールされているもの以外の特定の CA 証明書は必要ありません。

イベントストリーム処理のクライアントとサーバーは、ネゴシエートされた暗号化プロトコルと暗号スイートを TLSv1.2 に準拠させるように強制します。

TLS ハンドシェイクプロセスについて

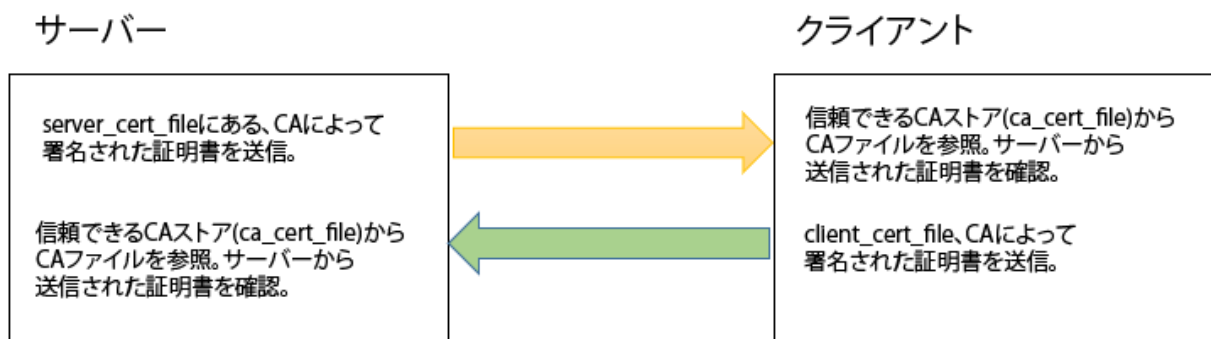
TLS ハンドシェイクプロセスは、サーバーと、必要に応じてクライアントによって設定された証明書を検証します。一方方向認証では、サーバーはクライアントによる証明書の送信を必要としません。クライアントは、サーバーから送信された証明書を検証します。

図 1 一方方向認証



双方向認証では、クライアントはサーバーによって検証される証明書も送信します。

図 2 双方向認証



自己署名証明書を受け入れたい場合は、trust_selfsigned に **true** を設定します。

注: 現在、C パブリッシュ/サブスクライブ API を使用してイベントストリーム処理サーバーに接続するクライアントのみが双方向認証を使用できます。

起動エラーがない場合、パブリッシュ/サブスクライブサーバーは、サーバーの起動時に記録される INFO レベルのログメッセージによって、TLS が有効か無効かを示します。TLS 用に有効になっている

クライアントは、接続を開始するとハンドシェイクステータスに関する INFO レベルのメッセージを記録します。クライアントまたはサーバーが TLS をネゴシエートしていて、そのピアがネゴシエートしていない場合、または TLS ハンドシェイク自体が失敗した場合、接続は失敗し、エラーメッセージが記録されます。

TCP/IP 接続での認証の有効化

概要

イベントストリーム処理エンジンに接続する TCP/IP クライアントの認証を要求することができます。認証を実装するには、次を手順を確認してください。

- OpenSSL ライブラリを ESP サーバーにインストールする必要があります
- ESP サーバーでパブリッシュ/サブスクライブ操作を有効にする必要があります

認証は、次のイベントストリーム処理エンジンの API に適用されます。

- パブリッシュ/サブスクライブ API
- ESP サーバーの HTTPS および WSS API
- イベントストリーム処理エンジンに接続するアダプタによって作成される接続
- C、Java、または Python パブリッシュ/サブスクライブ API を使用してイベントストリーム処理エンジンに接続する任意のクライアントによって作成される接続
- ESP クライアント(`dfesp_xml_client`)が HTTPS および WSS プロトコルを使用して ESP サーバーと通信するために作成する接続

サーバーで有効にすると、認証はグローバルかつ永続的な設定なので、そのサーバーに接続するすべてのクライアントを認証する必要があります。これは、パブリッシュ/サブスクライブ API によってサポートされているすべてのクライアント操作に適用されます。また、一意の TCP 接続を確立するクエリやその他のすべてのクライアント/サーバー要求にも適用されます。認証が失敗すると、クライアントは切断され、クライアントとサーバーの両方にエラーメッセージが記録されます。

同様に、認証が有効になっていないサーバーに対して認証を要求するクライアントも切断されます。対応するエラーメッセージがあります。

次の**セキュリティ構成プロパティ**を設定します。

```
security:
...
server:
  auth: null 1 # {null|oauth_local|saslogon|kerberos|oauth2}
...
client:
  auth: null # 1 {null|oauth_local|oauth2.password|oauth2.client_credentials|saslogon|kerberos}
...
```

- 1 auth は、**null** (認証を無効にするデフォルト値)、**oauth_local**、**saslogon**、**Kerberos**、**oauth2** という 5 つの可能な値のいずれかに設定します。スタンドアロン ESP サーバーおよびクライアントには

oauth_local を指定します。この認証方法は、既存のトークンと連動します。**oauth2** を使用するには、実行中の UAA サーバーが存在する必要があります。

注: ARM ベースのデバイスでは、OAuth2 クラウド認証のみがサポートされています。

ESP サーバーを **-auth** コマンドライン引数で起動すると、ESP サーバーのセキュリティ設定をオーバーライドできます。詳細は、[SAS Event Stream Processing: エッジ環境での ESP サーバーの使用](#)を参照してください。

次の 2 つのコマンドライン引数のいずれかを使用すると、クライアントの認証設定をオーバーライドできます。

- コネクタやアダプタなどのパブリッシュ/サブスクライブクライアントの URL を介して、認証の種類を指定します。URL の指定では、dfESP://host:port の後に **?auth_type** が必要です。このとき、**auth_type** には、**oauth_token**、**username**、**kerberos_servicename**、**grant_type** のいずれかの値を指定できます。
- **dfesp_xml_client** のコマンドラインでは **-auth mechanism** を指定します。

現在の認証メカニズムを表示するには、引数なしでコンソールに **client** コマンドを入力します。コマンドラインで認証メカニズムを設定しなければ、クライアントは構成ファイルを検索し、それを使用して認証構成を決定します。

OAuth トークン方式の認証

概要

注: OAuth トークン方式の認証は、ARM ベースのデバイスではサポートされていません。

OAuth トークン方式の認証メカニズムでは、OAuth 2.0/OpenID Connect 準拠のサーバーから取得した署名付き JSON Web トークンが必要です。このトークンは、ユーザーがパブリッシュ/サブスクライブ API またはアダプターに提供します。次に、トークンは、クライアントからサーバーへコンパクトなシリアル化形式(base64url 暗号化)で渡されます。これは、サーバーによって解析され、検証されます。

OAuth ローカル認証の場合、SAS Event Stream Processing では、Cloud Foundry (CF) User Account and Authentication (UAA)サーバーからトークンを取得する必要があります。CF UAA サーバーにトークンを要求し、そのトークンをクライアントに提供する必要があります。

curl を使用してローカルにインストールされた CF UAA サーバーへの REST 要求を呼び出すことによって、トークンを手動で要求できます。REST 要求は、認証情報フローで暗黙の許可を呼び出します。詳細については、CF UAA のマニュアルを参照してください。例については、“[CF UAA サーバー情報](#)”を参照してください。

次のリソースで詳細情報を確認できます。

- JSON Web トークン: <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 トークン方式のサーバー要件

エンジンを初期化するときにクライアント ID 文字列を渡すかセキュリティプロパティファイルを使用することで、イベントストリーム処理サーバー上で認証が有効になります。C++モデリング API では、これは `dfESPEngine::initialize()` 呼び出しのパラメーターです。認証を有効にするには、現在の `pubsub_ENABLE(portNum)` パラメーターを `pubsub_ENABLE_OAUTH(portNum, clientId)` に置き換えます。

サーバー上で OAuth ローカル認証を有効にするには、[セキュリティ構成プロパティ](#) を設定します。

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

- 1 `clientId` は、CF UAA サーバーからトークンを要求するときに使用される CF UAA `client_id` と一致する必要があります。詳細については、“[OAuth トークン検証](#)”を参照してください。
- 2 `key_file` には、`/opt/sas/viya/config/etc/SASEventStreamProcessingEngine/default/pubkey.pem` においてローカルファイルシステムでトークンの署名に使用された公開鍵を指定します。詳細については、“[OAuth トークン検証](#)”を参照してください。

トークンの検証でエラーが発生すると、サーバーはエラーコードをクライアントに返します。クライアントはサーバーから切断されます。

OAuth トークン方式のクライアント要件

クライアント上で OAuth ローカル認証を有効にするには、[セキュリティ構成プロパティ](#) を設定します。

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

クライアントは、トークンをサーバーに渡すことによって、認証された接続を要求します。トークンは、次のいずれかの方法で渡すことができます。

- 次のオプションエレメントを使用して、パブリッシュ/サブスクライブまたはアダプタ URL に渡します。

`?oauth_token`

このエレメントは、次のように URL の `host:port` 部分に続く必要があります。

dfESP://host:port?oauth_token=token... になります。URL の残りの部分は同じです。

- トークンを含むローカルファイルシステム上のファイルの完全なパスとファイル名を指定します。パブリッシュ/サブスクライブ API を使用する場合は、対応する C、Java、または Python のパブリッシュ/サブスクライブ API メソッドを呼び出します。C のメソッドは `C_dfESPpubsubSetTokenLocation()` で、Java と Python のメソッドは `setTokenLocation()` です。アダプターを実行するときは、対応するオプションのアダプター構成スイッチを使用してください。
- ESP クライアントの **-auth oauth-token** または **-auth oauth-token-url** コマンドラインパラメーターを使用します。

複数のメソッドを同時に使用することはできません。パブリッシュ/サブスクライブ API エラーが生成されます。

クライアントはトークンを隠ぺいされた状態でサーバーに渡し、トークンの検証結果を待ちます。正常に終了すると、接続が確立され、さらにクライアントサーバーの操作が正常に進行します。失敗した場合、クライアントはサーバーと接続を断ちます。

OAuth トークン検証

サーバーは、受信したトークン内の複数の項目を検証します。

検証項目	説明	
トークンシグネチャ	サーバーでは、OpenSSL ライブラリと <code>/opt/sas/viya/ config/etc/ SASEventStreamProcessingEngine/default/pubkey.pem</code> にある公開鍵を使用して、受信したトークンのシグネチャが検証されます。この公開鍵は、CF UAA サーバーで設定された公開鍵と秘密鍵のペアの公開鍵と同じ鍵でなければなりません。この鍵を生成し、セキュアにし、手動でサーバーにコピーする必要があります。詳細については、 “CF UAA サーバー情報” を参照してください。	
要求	aud	サーバー上で設定されたクライアント ID は、サーバーが受信したすべてのトークンに含まれる aud 要求と一致していなければなりません。トークン内の aud の値は、そのトークンを取得するための CF UAA サーバーへの要求に含まれる CF UAA クライアント ID によって決定されます。CF UAA 管理者は、イベントストリーム処理に特化したクライアント ID を設定する必要があります。認証が有効なサーバーを起動するときは、同じ ID を指定する必要があります。また、そのサーバーに接続するクライアントに対しトークンの使用を要求するときに、その ID を指定する必要があります。サーバー固有の特権は、そのサーバーへの接続にサーバー固有のクライアント ID を要求することによって適用できます。
	exp	トークンの有効期限が切れている場合は、トークン検証は失敗します。

CF UAA サーバー情報

CF UAA サーバーは、GitHub から入手可能なオープンソースのパッケージです。インストール後、次の追加の管理手順を強くお勧めします。

- OpenSSL (たとえば `openssl genrsa -out privkey.pem 1024`) を使用して新しい秘密鍵を生成し、その秘密鍵に基づいて公開鍵を生成します(`openssl rsa -pubout -in privkey.pem -out pubkey.pem`)。これらのキーをセキュアな状態に保ちます。
- CF UAA トークン検証鍵を公開鍵で、CF UAA トークン署名鍵を秘密鍵で設定します。次に、この CF UAA サーバーによって生成されたトークンを使用してクライアントを認証するすべてのイベントストリーム処理サーバーに公開鍵をコピーします。
- イベントストリーム処理サーバーまたはクライアントを実行しているユーザーのみが使用できるように制限された 1 つまたは複数の CF UAA クライアント ID を設定します。
- イベントストリーム処理クライアントが使用するトークンを必要とするユーザについて、CF UAA ユーザ名とパスワードの認証情報を登録します。

設定が終わったら、認証済み接続のトークンを取得して使用するために必要な手順は、次のとおりです。

- クライアントを特定のサーバーに接続するには、イベントストリーム処理サーバーで使用されているのと同じ公開鍵で設定された CF UAA サーバーからトークンを取得します。トークン要求には、サーバーで認証を有効にするために使用したのと同じ CF UAA クライアント ID が含まれている必要があります。CF UAA からトークンを要求する方法を選択できます。
- 応答からトークンを抽出し、“[OAuth トークン方式のクライアント要件](#)”で説明されている方法でクライアントに提供します。
- トークンが有効期限切れでない限り、同じサーバーに接続するときは、単一のトークンを無制限に再利用できます。

次に、認証情報を用いた暗黙的な付与により、CF UAA サーバーからトークンを取得するために curl を使用して呼び出した REST 要求の例を示します。

```
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"
```

REST 応答に成功した "302 Found" 応答が含まれている場合、REST 応答の Location フィールドの `&access_token` 部分にトークンを見つけることができます。

OAuth 2.0 クラウド認証および承認

OAuth 2.0 は、元の OAuth プロトコルに取って代わる業界標準のプロトコルです。OAuth 2.0 は、User Account and Authorization (UAA) サーバーでサポートされており、複雑な認証および承認サービスをサポートするためにクラウド配置で広く使用されています。次の情報は、通常はクラウドによってサービスとして提供される実行中の UAA サーバーの存在を前提としています。ESP サーバーおよびクライアントは、UAA サーバーと通信し、トークンを検証および取得する必要があります。

ESP サーバーでは、OAuth 2.0 の付与タイプ(リソース所有者のパスワードとクライアントの認証情報)がサポートされています。通常、ESP アダプタまたは REST クライアントは、リソースサーバー(この場合は ESP サーバー)にアクセスするためのクライアントとして機能します。リソースサーバーはクラウドに配置され、承認サーバー(Cloud Foundry で提供されるサーバーなど)によって保護されます。承認関連のタスクは、UAA サーバーに委任されます。このようにして、リソースサーバーはセキュリティ関連の情報にさらされることが少なくなり、したがって、セキュリティリスクの影響を受けにくくなります。

OAuth クラウド認証と承認を設定するには、適切な[セキュリティ設定プロパティ](#)を設定します。または、トークンを手動で要求し、OAuth ローカル認証にコマンドライン引数を使用することもできます。

ESP サーバーでは、次のセキュリティ構成プロパティを設定する必要があります。

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

- 1 UAA サーバーの管理者に連絡して `check_token_endpoint` を取得します。ここで **\$uaa.credentials.uri** の値を指定すると、ESP サーバーは JSON 形式を使用して Cloud Foundry が公開する VCAP 環境変数を解析しようとします。

注: リリース 2021.2.3 の時点で、SAS Event Stream Processing は [RFC 7622](#): トークンイントロスペクションをサポートしています。指定した URL の末尾に `/check_token` が自動的に追加されなくなりました。check_token エンドポイントを使用していて、それを引き続き使用する場合は、URL に `/check_token` を手動で追加する必要があります。その他の URL は、トークンイントロスペクションエンドポイントとして使用されます。

- 2 有効な `client_id` および `client_secret` を取得するには、UAA サーバーに登録します。

OAuth トークンが ESP サーバーによって受信されると、サーバーはその `client_id` および `client_secret` を使用して、トークンを UAA サーバーに転送します。UAA サーバーでは、REST コールによってこのトークンが発行され、サーバーへのアクセスが有効かつ承認済みかどうかチェックされました。

通常、UAA サーバーでは TLS が有効化されます(つまり、HTTPS プロトコルが使用されます)。UAA サーバー証明書が証明機関(CA)によって署名されている場合、システムのインストール済み CA 証明書は、ほとんどの場合、検証が可能です。ESP サーバーが自己署名している場合は、手動で証明書を取得し、`ca_cert` でその場所を指定することができます。

次のようにコマンドラインの **-auth** 引数を使用してプロパティをオーバーライドするという選択もできます。

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

サーバーと通信するには、各 ESP クライアントに有効な OAuth トークンが必要です。トークンを使用するには、次のセキュリティ構成プロパティを設定する必要があります。

```
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 クライアント(たとえば、アダプタおよび `dfesp_xml_client`)では、パスワードおよびクライアントの認証情報付与タイプがサポートされます。パスワード付与タイプを使用するには、UAA の有効なユーザー名、パスワード、クライアント ID、クライアントシークレットが必要です。クライアント認証情報付与タイプを使用するには、有効なクライアント ID とシークレットのみが必要です。

コマンドラインでクライアントを起動するときに、プロパティファイルをオーバーライドする引数を送信できます。クライアントで URL の指定がサポートされている場合、C++または Java アダプタの場合と同様に、URL は次のいずれかになります。

- `"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"`

付与タイプの後のパラメーターはオプションです。指定しない場合、ESP クライアントではプロパティファイルからパラメーター値が取得されます。クライアントで `-auth` コマンドライン引数がサポートされている場合は、次のいずれかを指定できます。

- `-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"`

前述のように、付与タイプの後のパラメーターはオプションです。

ユーザー名とパスワードを使用した認証(SASLogon サービス)

概要

注: この認証メカニズムは、ARM ベースのデバイスではサポートされていません。

この認証メカニズムでは、ユーザーが簡単なユーザー名とパスワードの認証情報を提供する必要があります。これらの認証情報は、イベントストリーム処理サーバーから SASLogon サービスに渡される時に有効である必要があります。

処理の順序は次のとおりです。

- 1 ユーザーは、パブリッシュ/サブスクライブ API、アダプター、または HTTP クライアントに認証情報を提供します。
- 2 認証情報は、変更されずにイベントストリーム処理サーバーに渡されます。
- 3 サーバーは、REST 要求における変更を加えていない認証情報を、構成済みの SASLogon サービスに渡します。
- 4 サーバーは、認証要求の結果をクライアントに返します。

SAS Event Stream Processing では、ユーザーが、少なくともパブリッシュ/サブスクライブ URL にユーザー名を含める必要があります。あるいは、パブリッシュ/サブスクライブ URL にパスワードを直接含めることもできます。それ以外の場合、イベントストリーム処理クライアント API は、クライアントのローカルファイルシステム内の **.authinfo** または **.netrc** ファイルを検索し、指定されたユーザー名と一致するパスワードを探します。どちらの方法でも、パスワードはクリアテキストまたは SAS でエンコードされたものを使用できます。

SASLogon サービスを使用した認証のサーバー要件

この認証方法は、エンジンを初期化するときに SASLogon サービスの URL を渡すことによって、ESP サーバーで有効になります。

構成ファイルを使用して認証メカニズムを設定するには、ESP サーバー上で次の[セキュリティ構成プロパティ](#)を指定します。

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

- 1 server_url に **\$saslogon_server** を指定した場合は、その環境変数の値が使用されます。

コマンドラインで ESP サーバーを起動するときは、**-auth saslogon://sasLogonURL** コマンドラインパラメーターを含めます。

SASLogon サーバーは HTTPS もサポートしています。

デフォルトでは、ESP サーバは SASLogon サーバーにサーバー証明書を要求します。有効な証明書が指定されていない場合、セッションは終了します。security.trust_selfsigned=true を設定すると、証明書が提供されていない場合や無効な証明書が提供されている場合でも、セッションは正常に進行します。

SASLogon サービスを使用した認証のクライアント要件

クライアントは、ユーザー名とパスワードをサーバーに渡すことによって、認証された接続を要求します。クライアントの[セキュリティ構成プロパティ](#)を指定します。

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

```
...
saslogon:
...
client:
  username: username
```

これらの値は、コマンドラインでクライアントを起動するときにオーバーライドできます。

- パブリッシュ/サブスクライブ URL またはアダプタ URL でオプションエレメント:?*username* および?*password* を使用して指定します。これらのエレメントは、URL の **host:port** 部分に続けて、**dfESP://host:port?username=username?password=password** のように記述する必要があります。URL の残りの部分は同じです。

パスワードはエンコードできますが、必ずしもそうである必要はありません。必要に応じて、PWENCODE プロシージャーを使用してパスワードをエンコードします。PWENCODE の結果のプレフィクス{SAS00x} を、イベントストリーム処理クライアントに渡されるパスワード文字列に含めるか、**.authinfo** ファイルまたは **.netrc** ファイルに含めるようにしてください。クライアントは、認証情報を隠べいされた状態でサーバーに渡し、SASLogon 認証の結果を待ちます。成功すると、接続が確立され、それ以降のクライアント/サーバーの操作は正常に進みます。失敗した場合、クライアントはサーバーから切断されます。

- ユーザー名は上で説明した方法で渡しますが、API にローカルの**.authinfo** または**.netrc** ファイルから一致するパスワードを抽出させます。この場合、環境変数 AUTHINFO または NETRC はフルパスとファイル名に設定する必要があります。環境変数が設定されていない場合、クライアント API はユーザーのホームディレクトリ内でファイルを検索します。
- ESP クライアントを起動するときは、**-auth saslogon://username** コマンドラインパラメーターを使用してユーザー名を渡します。

Kerberos 認証

概要

注: Kerberos 認証は、ARM ベースのデバイスではサポートされていません。

Kerberos 認証を有効にするには、ESP サーバーとクライアントが Kerberos サーバーにアクセスする必要があります。Kerberos サーバーは、サーバーとクライアントが使用するサービス名をサポートする必要があります。

これらの条件が満たされると、ユーザーは同じ Kerberos サービス名を ESP サーバーおよびクライアントに渡すことができます。クライアントが接続するとき、クライアントとサーバーの間に適切な Kerberos 交換が実行されます。クライアントは、ユーザーが渡したパブリッシュ/サブスクライブ URL に含まれているホストに、ユーザーが渡したサービス名を組み合わせ、サービスプリンシパルのフルネームを作成します。

Kerberos 認証のサーバー要件

イベントストリーム処理サーバーで Kerberos 認証を有効にするには、エンジンを初期化するときに Kerberos サービス名を渡す必要があります。C++モデリング API では、`dfESPEngine::initialize()`呼び出

しのパラメーターを使用してエンジンを初期化します。認証を有効にするには、現在の `pubsub_ENABLE(portNum)` パラメーターを `pubsub_ENABLE_KERBEROS(portNum, serviceName)` に置き換えます。

Kerberos サーバー環境には、読み取り可能な KeyTab ファイルが含まれている必要があります。このファイルは通常 `/etc/krb5.keytab` にあります。

ESP サーバーの次のセキュリティ構成プロパティを設定します。

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

これらの値は、**`-auth kerberos://serviceName`** コマンドラインパラメーターでオーバーライドできます。

Kerberos 認証のクライアント要件

クライアントは、サービス名をサーバーに渡すことによって、認証された接続を要求します。

Java および C++クライアントを含め、クライアントの次のセキュリティ構成プロパティを設定します。

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

これらの値は、**`-auth kerberos://serviceName`** コマンドラインパラメーターを使用してオーバーライドできます。サーバーは、Kerberos プロトコル交換を開始する前に、同じサービス名で構成されていることを検証します。

コマンドラインでクライアントを起動するときに、**`?kerberos_servicename`** エレメントを使用してサービス名を指定できます。このエレメントは、URL の **`host:port`** 部分に続く必要があります。たとえば、

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

URL の残りの部分は同じです。

クライアントを実行してローカルトークンキャッシュを更新する前に、**`kinit`** を実行する必要があります。

クライアントが Java クライアントの場合は、`jaas.conf` ファイルが SAS Event Stream Processing 構成ディレクトリに存在する必要があります。これは、すでにクライアントマシン上の他の Java クライアントによって使用されている `jaas.conf` ファイルのコピーである必要があります。ファイルには次の行が含まれている必要があります。

```
com.sun.security.jgss.krb5.initiate {
com.sun.security.auth.module.Krb5LoginModule required
  useTicketCache=true
  renewTGT=true
  doNotPrompt=true;
```

```
};
```

Java クライアントが認証に失敗した場合は、Java コマンドラインで次のデバッグプロパティを使用できます。

—**Dsun.security.krb5.debug=true -Dsun.security.jgss.debug=true**

デバッグ結果に “サポートされていないキータイプがデフォルトの TGT を見つけました: 18” のメッセージが表示された場合、Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy File をインストールする必要があります。

暗号化と認証の構成

次の YAML コードは、SAS Event Stream Processing での暗号化と認証を管理する `esp-properties.yml` 内のプロパティのサンプル値を示しています。

```
security:
#
# Encryption/Authentication Configuration for clients and Server
#
pubsub_ssl_enabled: false 1
http_ssl_enabled: false 2
2_way_authentication: false 3
trust_selfsigned: true 4
#
# Alternately, you can import the trusted CA/keystore/bundle certificate.
#
server:
  auth: null # {null|oauth_local|saslogon|kerberos|oauth2}
  server_cert_file: 5
  server_key_file: 6
  ca_cert_file: # specify when 2_way_authentication enabled
  passphrase: passphrase
client:
  auth: null # {null|oauth_local|oauth2.password|oauth2.client_credentials|saslogon|kerberos}
  ca_cert_file: 7
  client_cert_file: # specify when 2-way authentication is enabled
  passphrase:
#
# Authentication configuration parameters
#

# 1. Basic OAUTH2 authentication
#
oauth_local:
  server:
    client_id: clientid 8
    key_file: pubkey.pem
  client:
    token: null 9
    token_file: token.txt
# 2. Username/Password authentication via the SASlogon service
```

```
#
saslogon:
  server:
    server_url: URL
    #ca_cert: 10
  client:
    username: username
  http_proxy: null    # currently not supported
  https_proxy: null   # currently not supported
```

```
# 3. Kerberos authentication
```

```
#
kerberos:
  service_name: serviceName
```

```
# 4. OAuth2 (Cloud Foundry variant)
```

```
#
oauth2:
  server:
    check_token_endpoint: URL 11
  client:
    server_url: URL 12
    grant_types: 13
    password:
      username: name
      password: password
    client_credentials: null
  client_id: id
  client_secret: secret 14
```

- 1 パブリッシュ/サブスクライブのクライアントとサーバーは、TLS (Transport Layer Security)を有効にするかどうかをこのパラメーターを読み取って評価します。ここでは HTTP クライアントおよびサーバーと同じ証明書が使用されます。
 - 2 HTTP クライアントとサーバーは、TLS を有効にするかどうかをこのパラメーターを読み取って評価します。
 - 3 一方向 TLS 認証では、サーバーがそれが主張するサーバーであることをクライアントに証明する必要があります。ESP サーバーは、CA（認証局）によって署名された証明書をクライアントに送信します。この証明書がある場所を `server_cert_file` で指定します。サーバー証明書を受け取ったクライアントは、信頼できる CA ストアの CA ファイルを使用して、証明書が実際に CA によって署名されていることを確認します。クライアント CA ファイルの場所を `ca_cert_file` で指定します。サーバー証明書が正常に検証されると、TLS 接続を確立できます。
- 双方向認証では、クライアントがサーバーを認証した後、クライアント証明書（CA によって署名された）をサーバーに送り返す必要があります。この証明書がある場所を `client_cert_file` で指定します。サーバーは、`ca_cert_file` にある証明書を使用して検証します。
- 4 `true` に設定すると、証明書の検証をスキップします。SASLogon をサポートするには、こことセキュリティ構成ファイルの SASLogon セクションでこれを **true** に設定する必要があります。

注: Windows システムまたは Java クライアントでは `trust_selfsigned` は設定しません。Windows システムでは、信頼された CA 証明書ストアに CA 証明書をインポートします。Java クライアントでは、信頼されたキーストアに CA 証明書をインポートします。

- 5 証明書ファイルが存在する場合、その名前はデフォルトでは `server.pem` です。これは構成ディレクトリに置かれ、証明書と秘密鍵の両方が含まれている必要があります。
- 6 秘密鍵の名前です。存在しない場合は、`server_cert_file` に証明書と秘密鍵の両方が含まれていると仮定します。
- 7 このファイルが存在する場合、その名前はデフォルトでは `ca.pem` です。これは構成ディレクトリに置く必要があります。存在しない場合は、構成ディレクトリ内にインストールされている他の証明書が確認されます。
- 8 これは鍵ファイルで指定されたクライアント ID です。鍵ファイルの名前はデフォルトでは `pubkey.pem` です。User Account and Authentication Service (UAA) から ID を取得して、トークンシグネチャが署名されていることを確認します。
- 9 クライアントはまず、トークン自体がここで指定されているかどうかを確認します。そうでない場合、クライアントは構成ディレクトリ内の指定されたトークンファイルの存在を確認します。そのファイルが存在しない場合、クライアントは指定された URL にアクセスします。
- 10 このパラメーターは、CA 証明書の場所を指定します。
- 11 このパラメーターは、OAuth 2.0 クライアントがリソースサーバーに要求を送信するときにリソースサーバーがアクセストークンを確認する方法を指定します。このパラメーターは、環境変数と Cloud Foundry をサポートしています (例: `$uaa.credentials.uri`)。

注: リリース 2021.2.3 の時点で、SAS Event Stream Processing は [RFC 7622](#): トークンイントロスペクションをサポートしています。指定した URL の末尾に `/check_token` が自動的に追加されなくなりました。check_token エンドポイントを使用していて、それを引き続き使用する場合は、URL に `/check_token` を手動で追加する必要があります。その他の URL は、トークンイントロスペクションエンドポイントとして使用されます。

- 12 URL が `https://` で始まり、ESP サーバーが自己署名証明書を使用する場合、その証明書を Java クライアント用の信頼できるキーストアにインポートする必要があります。
- 13 これにより、リソース所有者のパスワードとクライアントの認証情報がサポートされます。
- 14 これは、`secret` を判断するために UAA 管理者とともに機能します。