日日是Oracle APEX

Oracle APEXを使った作業をしていて、気の付いたところを忘れないようにメモをとります。

2023年1月16日月曜日

OCI Vaultによる署名でJWTが生成できなかった件について

この前の記事でGoogleが提供しているREST APIを、サービス・アカウントを作成してアクセスしました。DBMS_CRYPTO.SIGNを使用しているため、Oracle Databaseは19c以上である必要があります。また、RSA鍵の扱いも気をつける必要があります。そのため、もう少し良い方法がないか考えてみました。

OCI VaultにGoogle側のサービス・アカウントの鍵をインポートし、Vaultで署名を行なうことによりJWTが作れないか試してみました。

結論としては、できませんでした。

JWTの署名アルゴリズムがRS256であれば、Vaultでの署名アルゴリズムとして**SHA_256_RSA_PKCS1_V1_5**を選択します。

Vaultを使った署名リクエストの定義を確認すると**message**の最大長は4096となっています。

https://docs.oracle.com/en-us/iaas/api/#/en/key/release/datatypes/SignDataDetails

しかし、以下のエラーが返されます。最大長は選択したアルゴリズム（SHA_256_RSA_PKCS1_V1_5を指定しています）に依存しているかもしれませんが、JWTのヘッダーとペイロードを署名するのに、245バイトでは足りません。

{"code":"InvalidParameter","message":"Message cannot be longer than 245 bytes. Message size = 284"}

そのため**messageType**に**DIGEST**を指定し、**message**としてSHA-256によるダイジェストの値をBase64でエンコードした文字列を指定する必要があります。

以下の処理でダイジェストの取得とBASE64でのエンコードを実施すればよい、と思ったのですが、処理に誤りがあるようです。

echo -n $MESSAGE | openssl sha256 -binary | base64

StackOverflowにCreate SHA256withRSA in two stepsという、同じ状況についての質問があります。この質問の回答と同様に、DigestInfoに含めずに、メッセージ・ダイジェストを直接署名していることが原因のようです。

うまくいっていませんが、行なった作業は記録しておきます。

RSAキーの形式変換

Googleのサービス・アカウントのキーの画面からは、PKCS#12形式でキーをダウンロードしています。

これをPKCS#1のPEM形式に変換します。最初にPKCS#8に変換します。

openssl pkcs12 -in <PKCS#12形式のファイル> -nocerts -nodes -out <出力ファイル>

出力するファイルは**pkey-pkcs8.pem**を指定しています。Enter Import Passwordとして**秘密のパスワード**を入力します。

```
% openssl pkcs12 -in my-first-api-project-******.p12 -nocerts -nodes -out pkey-pkcs8.pem
Enter Import Password: ********
MAC verified OK
%
```

続いてPKCS#1形式に変換します。

openssl rsa -in <PKCS#8形式のファイル> -out <PKCS#1形式のファイル>

-inには直前に実行したコマンドで出力したファイル名を与えます。今回の例では**pkey-pkcs8.pem**です。出力するファイルは**private.pem**としました。

```
% openssl rsa -in pkey-pkcs8.pem -out private.pem
writing RSA key
%
```

ここで作成したprivate.pemをOCI Vaultにインポートします。

非対称キーのインポート

非対称キーをインポートするには、パッチを適用したOpenSSLを作成する必要があります。

パッチの適用方法は、Oracle Cloud Infrastructureドキュメントの**非対称キーのインポート**として説明があります。

https://docs.oracle.com/ja-jp/iaas/Content/KeyManagement/Tasks/importing_assymetric_keys.htm#to_configure_and_patch_openssl

ドキュメントに書いてある通りの作業を行うためにAlways FreeのVM.Standard.E2.1.Microのインスタンスを作成します。そのインスタンスでOpenSSLへのパッチ適用とキー・マテリアルのラップを行います。

特別に作業ユーザーは作成せず、ユーザーopcを使用して作業します。

ドキュメントには1から10の作業が書かれています。ひとつひとつコピー＆ペーストを繰り返かえすことにより、エラーも発生することなく作業を完了できました。

```
[opc@mylinux ~]$ mkdir $HOME/build
[opc@mylinux ~]$ mkdir -p $HOME/local/ssl
```

```
[opc@mylinux ~]$ cd $HOME/build
[opc@mylinux build]$ openssl version
OpenSSL 1.1.1s  1 Nov 2022 (Library: OpenSSL 1.1.1k  FIPS 25 Mar 2021)
[opc@mylinux build]$ curl -O https://www.openssl.org/source/openssl-1.1.1d.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 8638k  100 8638k    0     0  36.8M      0 --:--:-- --:--:-- --:--:-- 36.8M
[opc@mylinux build]$ tar -zxf openssl-1.1.1d.tar.gz
[opc@mylinux build]$ sudo yum install patch make gcc -y
Failed to set locale, defaulting to C.UTF-8
Last metadata expiration check: 0:33:42 ago on Mon Jan 16 08:17:45 2023.
Package patch-2.7.6-11.el8.x86_64 is already installed.
Package make-1:4.2.1-11.el8.x86_64 is already installed.
Package gcc-8.5.0-16.0.1.el8_7.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[opc@mylinux build]$ cat <<-EOF | patch -d $HOME/build/ -p0
> diff -ur orig/openssl-1.1.1d/apps/enc.c openssl-1.1.1d/apps/enc.c
> --- orig/openssl-1.1.1d/apps/enc.c
> +++ openssl-1.1.1d/apps/enc.c
> @@ -533,6 +533,7 @@
>           */
>
>          BIO_get_cipher_ctx(benc, &ctx);
> +        EVP_CIPHER_CTX_set_flags(ctx, EVP_CIPHER_CTX_FLAG_WRAP_ALLOW);
>
>          if (!EVP_CipherInit_ex(ctx, cipher, NULL, NULL, NULL, enc)) {
>              BIO_printf(bio_err, "Error setting cipher %s\n",
> EOF
patching file openssl-1.1.1d/apps/enc.c
[opc@mylinux build]$ cd $HOME/build/openssl-1.1.1d/
[opc@mylinux openssl-1.1.1d]$ ./config --prefix=$HOME/local --
[省略]
[opc@mylinux openssl-1.1.1d]$ make -j$(grep -c ^processor /proc/cpuinfo)
[省略]
[opc@mylinux openssl-1.1.1d]$ make install
[省略]
[opc@mylinux openssl-1.1.1d]$ cd $HOME/local/bin/
[opc@mylinux bin]$
[opc@mylinux bin]$ echo -e '#!/bin/bash \nenv LD_LIBRARY_PATH=$HOME/local/lib/
$HOME/local/bin/openssl "$@"' > ./openssl.sh
[opc@mylinux bin]$ chmod 755 ./openssl.sh
[opc@mylinux bin]$ $HOME/local/bin/openssl.sh
OpenSSL> version
OpenSSL 1.1.1d  10 Sep 2019
OpenSSL> exit
[opc@mylinux bin]$
```

10の作業が終わるとOpenSSLが起動された状態になります。**version**を実行して1.1.1dであることを確認し、**exit**で終了します。

RSAキーのインポート

以下に記載されているスクリプトをkeywrap.shとして、OpenSSLをインストールしたインスタンスに作成します。

https://docs.oracle.com/ja-jp/iaas/Content/KeyManagement/Tasks/importing_asymmetric_keys_topic_script_to_import_rsa_key_material_as_a_new_external_key.htm

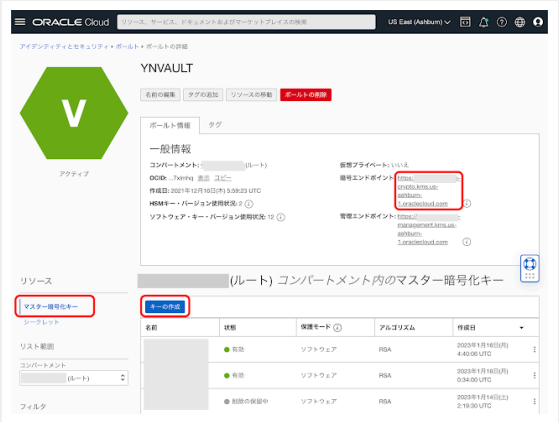
ファイルの先頭の指定は、以下のように変更します。

```
OPENSSL_PATH=$HOME/local/bin/openssl.sh
PRIVATE_KEY=$HOME/private.pem
WRAPPING_KEY=$HOME/public_wrapping_key.pem
```

すでに作成しているキー・ファイル**private.pem**を**/home/opc/private.pem**としてアップロードしておきます。

OCIコンソールの**ボールト**の画面より、**リソース**の**マスター暗号化キー**を開きます。暗号エンドポイントはAPIの呼び出し時に指定するため、メモっておきます。

キーの作成をクリックします。



画面右に、**キーの作成**を行うドロワーが開きます。

保護モードは**ソフトウェア**とします。**名前**を設定し（ここでは**BYOK RSA Key**）、**キーのシェイプ：アルゴリズム**として**RSA**、**キーのシェイプ：長さ**に**2048ビット**を選んでいます。これらの値はインポートするキーに合わせて指定を変えます。

鍵をインポートするため、**外部キーのインポート**に**チェック**を入れます。

公開ラッピング・キーに**カーソル**を当てます。

公開ラッピング・キーが表示されるので、コピーを実行しOpenSSLを準備したインスタンスに**/home/opc/public_wrapping_key.pem**というファイルとして作成します。

ファイルを作成したのち、**keywrap.sh**を実行します。**Binary wrapped key for console is available at**として、インポートできるファイルが作成されます。

```
[opc@mylinux ~]$ sh keywrap.sh
Openssl Path: /home/opc/local/bin/openssl.sh
Work Dir: /tmp/kms_WehX
Binary wrapped key for console is available at:
/tmp/kms_WehX/wrapped_key_material.bin
Base64 encoded wrapped key for CLI or API is available at:
/tmp/kms_WehX/wrapped_key_material.base64
[opc@mylinux ~]$
```

手元にダウンロードします。

```
% sftp -i mylinux.key opc@***.***.***.***
Connected to ***.***.***.***.
sftp> get /tmp/kms_WehX/wrapped_key_material.bin
Fetching /tmp/kms_WehX/wrapped_key_material.bin to wrapped_key_material.bin
wrapped_key_material.bin
100% 1736     5.0KB/s   00:00
sftp> exit
%
```

外部キーのデータ・ソースとして、ダウンロードした**wrapped_key_material.bin**を指定します。

キーの作成を実行します。

少々時間がかかりますが、**状態**が**有効**になればキーのインポートは完了です。

電子署名や検証には**マスター暗号化キーのOCID**を使うので、コピーを取っておきます。

OCI Vault側の作業は以上で完了です。

Web資格証明の準備

OCI VaultのREST APIを呼び出すユーザーは、以下の記事で作成した**apex_api_agent**を使用します。

APEXからOCIオブジェクト・ストレージを操作する(1) - APIユーザーの作成
http://apexugj.blogspot.com/2020/02/apex-with-oci-object-storage-1-apiuser.html

Web資格証明**OCI_API_ACCESS**の作成は以下の手順で実施します。

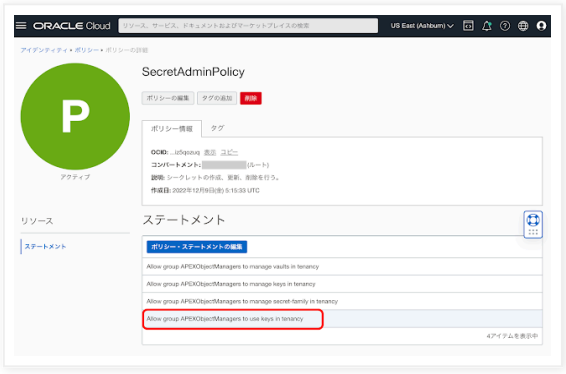
APEXからOCIオブジェクト・ストレージを操作する(4) - Web資格証明の作成
http://apexugj.blogspot.com/2020/02/apex-with-oci-object-storage-4-credential.html

ボールトを操作するためのポリシーの追加は、以下の記事の手順で実施します。

OCI PL/SQL SDKを使ってOCI Vaultのシークレットを操作する
http://apexugj.blogspot.com/2022/12/oci-secret-plsql-sdk.html

ポリシーについては、以下も追加します。

Allow group APEXObjectManagers to use keys in tenancy



OCI VaultのREST APIを呼び出すまでの準備は、以上で完了です。

動作確認

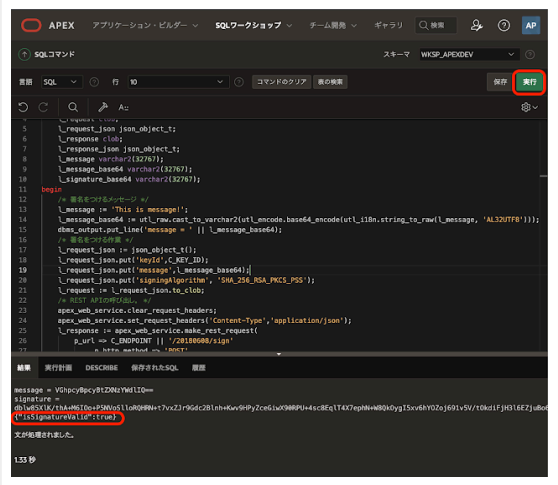
以下のコードを実行し、メッセージの署名と検証を行います。**C_ENDPOINT**として**暗号エンドポイント**、**C_KEY_ID**としてインポートした**マスター暗号化キーのOCID**を指定します。

```
declare
    C_ENDPOINT constant varchar2(200) := '暗号エンドポイント';
    C_KEY_ID   constant varchar2(200) := 'マスター暗号化キーのOCID';
    l_request clob;
    l_request_json json_object_t;
    l_response clob;
    l_response_json json_object_t;
    l_message varchar2(32767);
    l_message_base64 varchar2(32767);
    l_signature_base64 varchar2(32767);
begin
    /* 署名をつけるメッセージ */
    l_message := 'This is message!';
    l_message_base64 := utl_raw.cast_to_varchar2(utl_encode.base64_encode(utl_i18n.string_to_ra
    dbms_output.put_line('message = ' || l_message_base64);
    /* 署名をつける作業 */
    l_request_json := json_object_t();
    l_request_json.put('keyId',C_KEY_ID);
    l_request_json.put('message',l_message_base64);
    l_request_json.put('signingAlgorithm', 'SHA_256_RSA_PKCS1_V1_5');
    l_request := l_request_json.to_clob;
    /* REST APIの呼び出し。 */
    apex_web_service.clear_request_headers;
    apex_web_service.set_request_headers('Content-Type','application/json',p_reset => false);
    l_response := apex_web_service.make_rest_request(
        p_url => C_ENDPOINT || '/20180608/sign'
            ,p_http_method => 'POST'
            ,p_body => l_request
            ,p_credential_static_id => 'OCI_API_ACCESS'
    );
    l_response_json := json_object_t(l_response);
    l_signature_base64 := l_response_json.get_string('signature');
    dbms_output.put_line('signature = ' || l_signature_base64);
    /* 署名の検証を行う */
    l_request_json := json_object_t();
    l_request_json.put('keyId',C_KEY_ID);
    l_request_json.put('message',l_message_base64);
    l_request_json.put('signature',l_signature_base64);
    l_request_json.put('signingAlgorithm', 'SHA_256_RSA_PKCS1_V1_5');
    l_request := l_request_json.to_clob;
    /* REST APIの呼び出し。 */
    apex_web_service.clear_request_headers;
    apex_web_service.set_request_headers('Content-Type','application/json',p_reset => false);
```

```
    l_response := apex_web_service.make_rest_request(
        p_url => C_ENDPOINT || '/20180608/verify'
            ,p_http_method => 'POST'
            ,p_body => l_request
            ,p_credential_static_id => 'OCI_API_ACCESS'
    );
    dbms_output.put_line(l_response);
end;
```

test_sign_verify_valut.sql hosted with ❤ by GitHub view raw

SQLワークショップのSQLコマンドより実行します。



verifyを呼んだ時の応答として、trueが返ってきていれば検証は成功です。

{"isSignatureValid":true}

今回の記事は以上になります。

完

Yuji N. 時刻: 19:20

共有

自己紹介

Yuji N.

日本オラクル株式会社に勤務していて、Oracle APEXのGroundbreaker Advocateを拝命しました。
こちらの記事につきましては、免責事項の参照をお願いいたします。

詳細プロフィールを表示