JWE-JWS User Guide

User Guide for creating, verifying and decrypting JWE-JWS

Version 1.0.0

Visa Confidential

Revision History

Version	Date	Comments	Created/Updated By
1.0.0	July 5	Initial version	Purnachandra Motati (pmotati@visa.com) Shameem Peerbuccoss (speerbuc@visa.com)

Table of Contents

Introduction

This document describes the encryption mechanism the VISA uses to encrypt and decrypt sensitive data such as PAN and address. Field Level Encryption (FLE) is used for this purpose both on the incoming messages from the client to Visa and outgoing messages from Visa to the client.

All the data elements in the request and response payloads that have the prefix "enc" are encrypted.

The following VISA products, not limited to, use the JWE & JWS encryption where it us applicable:

- Visa Direct (VD) – Message Level Encryption (MLE)
- Visa In-App Provisioning (VIAP)
- Visa Token Service (VTS)
- Visa Token Service Provisioning (VTIS) and Lifecycle Management (LCM)
- Visa Installment (VI)
- Visa Acceptance Cloud (VAC)

Visa uses JSON Object Signing and Encryption (JOSE) technologies – JSON Web Signature [JWS] and JSON Web Encryption (JWE) to encrypt and/or sign content using a variety of algorithms. Encrypted input parameters must be constructed before sending them in API requests.

JWE – JSON Web Encryption

The JWE specification standardizes the way to represent encrypted content in a JSON-based data structure.

See https://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-40 for more details and complete specifications for JWE.

Visa supports both symmetric (JWE using API Key / Shared Secret) and asymmetric algorithms (JWE using RSA PKI).

For symmetric keys, the corresponding key ID / API Key and shared secret are assigned to the client during onboarding where the shared secret will be generated and provided by Visa.

For asymmetric keys, during onboarding both Visa and the client would share their public keys. Client will encrypt all necessary fields in the request payload using Visa's public key and Visa will decrypt these fields using its corresponding private key. Similarly, Visa will encrypt all

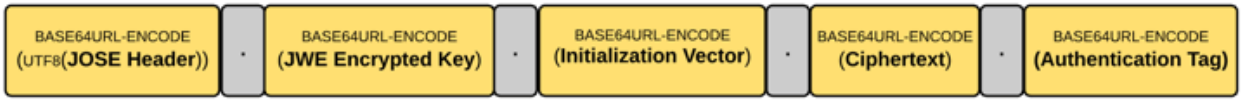
necessary fields in the response payload using the client's public key and client will decrypt these fields using their private key.

Refer to the API Specification document for more information.

JWE Composition

Visa uses the Compact serialization scheme for JWE. The JWE composition is defined as follows:

Base64URL (UTF8 (JWE Header)) || '.' || Base64URL (JWE Encrypted Key) || '.' || Base64URL (JWE Initialization Vector) || '.' || Base64URL (JWE Ciphertext) || '.' || Base64URL (JWE Authentication Tag)



Base64 encoding & conventions that Visa follows:
Visa has chosen to suffix ".." to the Base-64 encode values, instead of the more popular == convention.

JWE Header

The JWE header contains the following metadata for encryption and decryption.

Table below shows the common JWE header that are required.

Name	Description	VIAP	VTS VTIS LCM	VI	VD
alg	Description of the algorithm used to encrypt the randomly generated symmetric Content Encryption Key (CEK). **Format**: Values are AGCM256KW (Symmetric), RSA-OAEP-256 (Asymmetric).	✓	✓	✓	✓
typ	Media type of the JWE. JavaScript Object Signing and Encryption (JOSE) indicates that JWE is using JWE Compact Serialization scheme. **Format**: Value is JOSE.	✓	✓	✓	✓

kid	Key identifier that identifies the key used to encrypt the CEK, which is the Issuer's API key. **Format**: String; max length 64 character	✓	✓	✓	✓
enc	Type of encryption used by the CEK to encrypt sensitive payload elements. **Format**: Values are A128GCM and A256GCM.	✓	✓	✓	A128GCM

Visa also requires an additional set of header fields to be provided. Refer to the corresponding API specification document to determine which header(s) are applicable.

✓ = Required, X = No Required

O = Optional, C= Conditional

Name	Description	VTS	VTIS	VIAP	VI	VD
iat[2]	Time when JWE was issued. Expressed in UNIX epoch time (seconds since 1 January 1970) and issued at timestamp in UTC when the transaction was created and signed. **Format**: yyyy-MM-ddTHH:mm:ss.SSSZ.	VCEH[1], PP[2]	✓	C	X	O[3]
exp	Expiration time of the payload. This field is optional and should be evaluated in conjunction with the ttl default defined with use case. exp can be used to reduce the default ONLY. If exp represents a time greater than iat + ttl then iat + ttl will be used. **Format**: yyyy-MM-ddTHH:mm:ss.SSSZ.	VCEH[1]	O	C	X	X
iss	Message originator client ID, which is the vClientID of the issuer. The field is required for pushing card enrollment to wallet providers. The message originator is the issuer. **Format**: String; alphanumeric; max length 64 characters.	VCEH[1]	C	X	X	X
aud	List of desired recipients, which are the token requestors' vClientID values. This field is required for pushing card enrollment to	X	C	X	X	X

	wallet providers. The recipients are token requestors. The value should be a comma-separated array of client IDs. Up to 10 client IDs are allowed for each request. **Format**: String; alphanumeric; max length 256 characters.					
channel Security Context	This indicates which channel security to use for encryption & decryption Values: RSA_PKI, OPACITY_A, OPACITY_B, SHARED_SECRET	✓	✗	✗	✗	✗
jti	The "jti" (JWT ID) claim provides a unique identifier for the JWT.	✗	✗	✗	✗	✗

1. VCEH -Visa Card Enrollment Hub
2. PP – Push Provisioning. iat is mandatory for push provisioning. it is also conditionally required when the exp header is present.
3. VD – iat is not required for P2W (Push to Wallet) and P2A (Push to Account)

JWE Encrypted Key

The JWE Encrypted Key is a randomly generated Content Encryption Key (CEK).
Note:

- AGCM256KW algorithm should be used for Symmetric
- RSA-OAEP-256 algorithm should be used together with the receiver's public encryption key

JWE Initialization Vector

The JWE Initialization Vector (IV) is a randomly generated initialization vector (also known as salt) of 96 bits length in Base 64 URL Safe encoded form.

JWE Ciphertext

JWE Ciphertext is an encrypted BLOB that is generated from the plaintext (sensitive data that needs to be encrypted), by using the A256GCM encryption scheme specified in the enc header field.

JWE Authentication Tag

An HMAC (hash-based message authentication code) encryption of 128 bits is generated from the plaintext authentication tag using the A256GCM algorithm (A256GCM).

Creating JWE using API Key / Shared Secret

The encryption key provided during onboarding is used to encrypt and decrypt payloads. JSON Web Encryption (JWE) content should be signed or encrypted using the shared secret that was provided to client at the time of onboarding.

Below are the symmetric encryption parameters:

- alg: 'AGCM256KW'
- typ: 'JOSE'
- tag: '<128-bit HMAC generated from applying AES-256-GCM-KW to the CEK>'
- kid: '<APIKey>'
- enc: '<Encryption Algorithm to be used.>'
- iv: '<A unique 96-bit Base64 encoded value>'

Note:

The JWE Protected Header is input as the AAD (Additional Authenticated Data) parameter of the authenticated encryption (AES-GCM) of the "text to encrypt".

The general approach for JWE Encryption using API Key / Shared Secret are as follows:

- Visa uses the compact serialization style. (elements separated by ".").
- All fields are base 64 – URL Safe encoded with NO padding.
- Visa will use a CEK – Content Encryption Key – 256bits size.
- You must use an AES-GCM-256KW algorithm and an Initialization Vector (IV) for encryption of content encryption key. Size of IV is to be 96 bits
- Authentication Tag will be generated as an additional output of the AES-GCM-256 encryption. Size of this field is 128 bits.
- All string-to-byte and vice-versa conversions are done with UTF-8 charset

General steps for encrypting the plain text data.

1. Get the shared secret for the API key.
2. Create a SHA-256 digest of the shared secret.
3. Generate a random Content Encryption Key (CEK)
4. Encrypt the CEK for the recipient using the shared secret digest (32-byte value), the 96-bit random IV, and the algorithm AGCM256-KW specified in alg element. Base64UrlSafe encode to produce an Encrypted-KEY.
5. Generate a random IV of length 96 bit. Base64-UrlSafe encode to produce Payload-IV.
6. Encrypt plaintext data using the CEK and Payload-IV, and the algorithm AGCM256 specified in the enc section to form the ciphertext and the Payload Tag data.

7. Base64-UrlSafe encode the encrypted bytes to produce ciphertext.
8. Base64-UrlSafe encode the Tag data to produce TAG.
9. Base64-UrlSafe encode the entire JWE Header JSON containing the encryption parameters used in clear text as shown above, to produce the encoded header.

Sample JWE Header sample for Symmetric Encryption:

```
{
  "alg": "AGCM256KW",
  "typ": "JOSE",
  "tag": "<128bitvalue>", //HMAC generated from applying AES-256-GCM-KW to the CEK
  "kid": "<APIKey>", //API Key
  "enc": "A256GCM"
}
```

Sample JWE Body

- encrypted_key: base64 encoded form. CEK encrypted using AGCM256KW (alg) algorithm and the CEK IV
- iv: base64 encoded form. IV for the text encryption. Size of IV is to be 96-bit Base64 encoded form
- ciphertext: encrypted blob generated using the AES-GCM encryption (enc) of the text to encrypt
- tag: base64 encoded form. HMAC generated using the AES-GCM encryption of the text to encrypt. The size of the tag should be 128 bits

```
"encrypted_key": "UghIOgu ... MR4gp_A=",
"iv": "AxY8DctDa....GlsbGljb3RoZQ=",
"ciphertext": "KDlTthhZTGufMY.....xPSUrfmqCHXaI9wOGY=",
"tag": "Mz-VPPyU4...RlcuYv1IwIvzw="
```

Decrypting a JWE using Shared Secret

1. Base64-UrlSafe decode the E-Header field.
2. Get the kid identifier (API Key) and the algorithms to use for decryption.
3. Fetch the ding keys for ECDH operation and perform ECDH and KDF operation to arrive at the Secret key
4. Base64-UrlSafe decode the E-Authentication Tag for use in ciphertext decryption.
5. Base64-UrlSafe decode the encoded IV for use in ciphertext decryption.
6. Base64-UrlSafe decode the encoded CipherText field to get the ciphertext.
7. Decrypt the cipher text using Secret key, and authentication Tag and encryption algorithm as specified in alg element (AGCM256).

Java Sample Code

Visa uses the JOSE Nimbus Library for JWE. Below is the maven dependency that needs to be added to the POM file of your project.

Note: Ensure that the version you are using is approved by your security team.

Maven Dependency

```
<dependencies>
    <dependency>
        <groupId>org.bouncycastle</groupId>
        <artifactId>bcprov-jdk16</artifactId>
        <version>1.46</version>
    </dependency>
    <dependency>
        <groupId>com.nimbusds</groupId>
        <artifactId>nimbus-jose-jwt</artifactId>
        <version>9.1.2</version>
    </dependency>
</dependencies>
```

Sample Code:

```
/*
 *
 *   © Copyright 2018 - 2020 Visa. All Rights Reserved.
 *
 *   NOTICE: The software and accompanying information and documentation
(together, the "Software") remain the property of and are proprietary to
Visa
 *   and its suppliers and affiliates. The Software remains protected by
intellectual property rights and may be covered by U.S. and foreign patents
or patent applications.
 *   The Software is licensed and not sold.
 *
 *  By accessing the Software you are agreeing to Visa's terms of use
(developer.vis.com/terms) and privacy policy (developer.visa.com/privacy).
 *  In addition, all permissible uses of the Software must be in support of
Visa products,
 *  programs and services provided through the Visa Developer Program (VDP)
platform only (developer.visa.com).
 *  **THE SOFTWARE AND ANY ASSOCIATED INFORMATION OR DOCUMENTATION IS
PROVIDED ON AN "AS IS," "AS AVAILABLE," "WITH ALL FAULTS" BASIS WITHOUT
WARRANTY OR  CONDITION OF ANY KIND. YOUR USE IS AT YOUR OWN RISK.**
 *  All brand names are the property of their respective owners, used for
identification purposes only,
 *  and do not imply product endorsement or affiliation with Visa. Any
links to third party
 *  sites are for your information only and
 *  equally  do not constitute a Visa endorsement. Visa has no insight into
and control over
 *  third party content and
```

```
 *  code and disclaims all liability for any such components, including
continued availability
 *  and functionality.
 *  Benefits depend on implementation details and business factors and
coding steps shown are exemplary only and
 *  do not reflect all necessary elements for the described capabilities.
Capabilities and
 *  features are subject to Visa's terms and conditions and
 *  may require development, implementation and resources by you based on
your business
 *  and operational details.
 *  Please refer to the specific API documentation for details on the
requirements, eligibility
 *  and geographic availability.
 *
 *  This Software includes programs, concepts and details under continuing
development by
 *  Visa. Any Visa features, functionality, implementation, branding, and
 * schedules may be amended, updated or canceled at Visa's discretion.
 *  The timing of widespread availability of programs and functionality is
also subject to a number of factors outside Visa's control, including but
 *  not limited to deployment of necessary infrastructure by issuers,
acquirers, merchants
 *  and mobile device manufacturers.
 *
 */

import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.AESDecrypter;
import com.nimbusds.jose.crypto.AESEncrypter;
import com.nimbusds.jose.crypto.MACSigner;
import com.nimbusds.jose.crypto.MACVerifier;
import com.nimbusds.jwt.EncryptedJWT;

import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
import java.util.Map;

/**
 * @author speerbuc@visa.com
 **/
public final class JWEUtilsUsingSharedSecret {

    private JWEUtilsUsingSharedSecret() {
    }

    /**
     * Create JWE Using Shared Secret
     *
     * @param plainText    - Plain Text to encrypt
     * @param apiKey       - Key ID (API Key)
     * @param secretSecret - Shared Secret
     * @return JWE String in Compact Serialization Format
     * @throws Exception
     */
    public static String createJwe(String plainText, String apiKey, String
sharedSecret) throws Exception {
```

```
        JWEHeader.Builder headerBuilder = new
JWEHeader.Builder(JWEAlgorithm.A256GCMKW, EncryptionMethod.A256GCM);
        headerBuilder.keyID(apiKey);
        headerBuilder.customParam("iat", System.currentTimeMillis());

        JWEObject jweObject = new JWEObject(headerBuilder.build(), new
Payload(plainText));
        jweObject.encrypt(new AESEncrypter(sha256(sharedSecret)));
        return jweObject.serialize();
    }

    /**
     * Decrypt JWE with Shared Secret
     *
     * @param jwe          - JWE String in Compack Serilaization Form
     * @param sharedSecret - Shared Secret
     * @return - Plain Text
     * @throws Exception
     */
    public static String decryptJwe(String jwe, String sharedSecret) throws
Exception {
        EncryptedJWT encryptedJWT = EncryptedJWT.parse(jwe);
        encryptedJWT.decrypt(new AESDecrypter(sha256(sharedSecret)));
        return encryptedJWT.getPayload().toString();
    }

    /**
     * Create A SHA256 hash of the input
     *
     * @param input - String value
     * @return - SHA-256 hash in bytes
     * @throws NoSuchAlgorithmException
     */
    private static byte[] sha256(String input) throws
NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        md.update(input.getBytes(StandardCharsets.UTF_8));
        return md.digest();
    }
}
```

Creating JWE Using RSA PKI

The JWE content should be encrypted using the public encryption key generated by the client and provided (in a certificate in PEM format) to Visa during onboarding.

Below are the asymmetric encryption parameters:

- alg: 'RSA-OAEP-256'
- typ: 'JOSE'
- kid: '<APIKey>'
- enc: 'A256GCM'

The general approach for encrypting data using JWE and RSA PKI is as follows:

- Visa uses the compact serialization style, wherein elements are separated by a "."(period).
- All fields are Base64URL-encoded.
- Visa uses the hybrid encryption scheme. In this method, an RSA 2048 key is used to encrypt a random symmetric key. The random symmetric key is then used to encrypt the text.
- Use the RSA-OAEP-256 algorithm, encrypting the random symmetric key.
- Use the A256GCM algorithm for the encryption of text with an Initialization Vector (IV). Size of IV should be 96 bits.
- Authentication Tag will be generated as an additional output of the A256GCM encryption. Size of this field is 128 bits.
- JWE header is used to pass AAD (Additional Authentication Data) for the A256GCM operation.
- All string-to-byte and byte-to-string conversions are done with UTF-8 charset.

General steps for encrypting the plain text data.

Apply JWE on plaintext data and encrypt data by following these steps:

1. Get the RSA 2048 public key.
2. Generate a Random Symmetric Key (RSK) of 256 bits length.
3. Encrypt the RSK that is using the RSA 2048 key, by using the algorithm specified in alg (RSA-OAEP-256).
4. Generate a random IV of 96 bits length and perform Base64URL encoding to produce E-IV.
5. Encrypt plaintext data, by using the RSK, Payload-IV, and the algorithm A256GCM specified in the enc header field to form the ciphertext and the payload tag data.
6. Base64URL-encode the encrypted bytes to produce E-Ciphertext.
7. Base64URL-encode the Tag data to produce E-TAG.
8. Base64URL-encode the RSK to produce E-Encrypted Key.

9. Base64URL-encode the entire JWE header JSON containing the encryption parameters used in clear text as shown below, to produce the JWE.

Sample JWE Header sample for Asymmetric Encryption:

```
{
"alg":"RSA-OAEP-256",
"kid":"<API KEY>",
"typ":"JOSE",
"enc":"A256GCM",
"iat":"1429837145"
}
```

Sample JWE Body

- encrypted_key: base64 encoded form. CEK encrypted using A256GCM (alg) algorithm and the CEK IV
- iv: base64 encoded form. IV for the text encryption. Size of IV is to be 96-bit Base64 encoded form
- ciphertext: encrypted blob generated using the AES-GCM encryption (enc) of the text to encrypt
- tag: base64 encoded form. HMAC generated using the AES-GCM encryption of the text to encrypt. The size of the tag is to be 256 bits

```
"encrypted_key": "UghIOgu ... MR4gp_A=",
"iv": "AxY8DctDa....GlsbGljb3RoZQ=",
"ciphertext": "KDlTthhZTGufMY....xPSUrfmqCHXaI9wOGY=",
"tag": "Mz-VPPyU4...RlcuYv1IwIvzw="
```

Decrypting a JWE Using RSA

1. Get the RSA 2048 private key.
2. Decode the Base64URL E-Header field.
3. Get the kid and the algorithms to use for decryption.
4. Decode the Base64URL E-Key field to get the encrypted E-key (Random Symmetric Key).
5. Decrypt the JWE Encrypted Key field by using the RSA 2048 private key with the encryption algorithm, as specified in the alg element (RSA-OAEP-256).
6. JWE header (the base64 encoded string in bytes) is used to pass AAD (Additional Authentication Data) for the A256GCM operation.
7. Get the RSK in clear.

8. Decode the Base64URL E-IV for use in ciphertext decryption.
9. Decode the Base64URL E-Authentication Tag for use in ciphertext decryption.
10. Decode the Base64URL ciphertext field.
11. Decrypt the ciphertext, by using RSK and IV and authentication tag and encryption algorithm, as specified in the enc header field (A256GCM).

Java Sample Code

Visa uses the JOSE Nimbus Library for JWE. Below is the maven dependency that needs to be added to the POM file of your project.

Note: Ensure that the version you are using is approved by your security team.

Maven Dependency

```
<dependencies>
    <dependency>
        <groupId>org.bouncycastle</groupId>
        <artifactId>bcprov-jdk16</artifactId>
        <version>1.46</version>
    </dependency>
    <dependency>
        <groupId>com.nimbusds</groupId>
        <artifactId>nimbus-jose-jwt</artifactId>
        <version>9.1.2</version>
    </dependency>
</dependencies>
```

Sample Code

```
import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.AESDecrypter;
import com.nimbusds.jose.crypto.RSADecrypter;
import com.nimbusds.jose.crypto.RSAEncrypter;

import java.io.File;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.security.GeneralSecurityException;
import java.security.KeyFactory;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.text.ParseException;
import java.util.Base64;
import java.util.Date;

public class JWEUtilsUsingRSAPKI {
```

```
    /**
     * Create JWE using RSA Public Key
     *
     * @param data       - Plain Text
     * @param rsaPubKey - RSA Public Key
     * @param kid        - Key UserId
     * @return JWE String in compact serialization format
     * @throws Exception
     */
    public static String createJwe(String data, RSAPublicKey rsaPubKey,
String kid) throws Exception {
        long currentTime = (new Date()).getTime() / 1000L;
        JWEHeader updatedHeader = (new
JWEHeader.Builder(JWEAlgorithm.RSA_OAEP_256, EncryptionMethod.A256GCM))
                .keyID(kid).type(JOSEObjectType.JOSE).customParam("iat",
currentTime).build();
        JWEObject jweObject = new JWEObject(updatedHeader, new
Payload(data));
        RSAEncrypter encrypter = new RSAEncrypter(rsaPubKey);
        jweObject.encrypt(encrypter);
        return jweObject.serialize();
    }

    /**
     * Decrypt JWE Using RSA PKI
     *
     * @param jwe           - JWE String in compact serialization format
     * @param privateKeyPem - RSA Private Key in PEM Format
     * @return Plain Text
     * @throws GeneralSecurityException
     * @throws ParseException
     */
    public static final String decryptJwe(String jwe, String privateKeyPem)
throws GeneralSecurityException, ParseException {
        String privateKey = privateKeyPem
                .replace("-----BEGIN PRIVATE KEY-----", "")
                .replaceAll(System.lineSeparator(), "")
                .replace("-----END PRIVATE KEY-----", "");

        byte[] encodedPrivateKey = Base64.getDecoder().decode(privateKey);

        String plainText;
        JWEObject jweObject = JWEObject.parse(jwe);
        JWEHeader header = jweObject.getHeader();
        JWEAlgorithm jweAlgorithm = header.getAlgorithm();
        try {
            if (JWEAlgorithm.RSA1_5.equals(jweAlgorithm) ||
JWEAlgorithm.RSA_OAEP_256.equals(jweAlgorithm)) {
                RSAPrivateKey rsaPrivateKey = (RSAPrivateKey)
KeyFactory.getInstance("RSA").generatePrivate(new
PKCS8EncodedKeySpec(encodedPrivateKey));
                RSADecrypter decrypter = new RSADecrypter(rsaPrivateKey);
                jweObject.decrypt(decrypter);
                plainText = jweObject.getPayload().toString();
            } else {
                JWEDecrypter decrypterForAES = new
AESDecrypter(encodedPrivateKey);
                jweObject.decrypt(decrypterForAES);
                plainText = jweObject.getPayload().toString();
            }
        } catch (JOSEException e) {
```

```
            throw new GeneralSecurityException("JOSEException has
encountered.", e);
        }
        return plainText;
    }

    /**
     * Load RSA Public Key From File (PEM Format)
     *
     * @param publicKeyFile - Public Key File
     * @return {@link RSAPublicKey}
     * @throws Exception
     */
    public static RSAPublicKey loadPublicKeyFromPemFile(File publicKeyFile)
throws Exception {
        String key = new String(Files.readAllBytes(publicKeyFile.toPath()),
Charset.defaultCharset());

        String publicKeyPEM = key
                .replace("-----BEGIN PUBLIC KEY-----", "")
                .replaceAll(System.lineSeparator(), "")
                .replace("-----END PUBLIC KEY-----", "");

        byte[] encoded = Base64.getDecoder().decode(publicKeyPEM);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encoded);
        return (RSAPublicKey) keyFactory.generatePublic(keySpec);
    }

}
```

JWS (JSON Web Signature)

The JSON Web Signature (JWS) is a compact signature format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. The JWS signature mechanisms are independent of the type of content being signed, allowing arbitrary content to be signed. A related encryption capability is described in a separate JSON Web Encryption (JWE) [JWE] specification.

Please refer to https://tools.ietf.org/html/rfc7515 for more details.

JWS Compositions

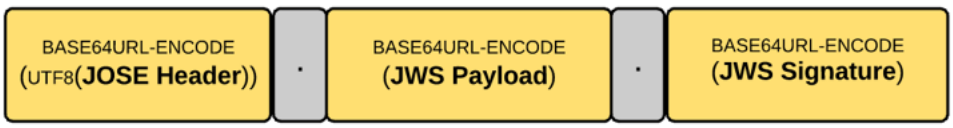
JWS represents signed content using JSON data structures and base64url encoding. The representation consists of three parts:

- the JWS Header,
- the JWS Payload,
- and the JWS Signature.

The three parts are base64url-encoded for transmission, and typically represented as the concatenation of the encoded strings in that order, with the three strings being separated by period ('.') characters.

The JWS Header describes the signature method and parameters employed. The JWS Payload is the message content to be secured. The JWS Signature ensures the integrity of both the JWS Header and the JWS Payload.

Base64URL (UTF-8 (JWS Header)) || '.' || Base64URL (JWS Payload) || '.' || Base64URL (JWS Signature)



Base64 encoding & conventions that Visa follows:

Visa has chosen to suffix ".." to the Base-64 encode values, instead of the more popular == convention.

JWS Header

The JWE header contains the metadata for encryption and decryption.

Name	Description
alg	(*Required*) A description of the algorithm that is used to sign the JWS payload **Format**: Value is PS256 (RSA PKI) or HS256 (Shared Secret).
Kid	(*Required*) The key identifier that identifies the key used to sign the JWS payload. **Format**: String; max length 64 characters.
typ	(*Required*) The media type of the JWS. JOSE indicates that the JWS is using the JWS Compact Serialization scheme. **Format**: Value is JOSE.
cty	(Optional) The content type of the JWS payload. **Format**: Value is JWE.

JWS Payload

The JWS payload is the payload being signed.

- Payload must be either in Base64 encode plain text format or it should be JWE payload.
- All Issuer inbound APIs to VISA from the Issuers (Banks) must be JWE payload only.

JWS Signature

The JWS signature is generated to protect the integrity of both the JWS header and JWS payload.

JWS Using Shared Secret

Java Sample Code:

```
/**
 * Create JWS Using Shared Secret
 *
 * @param jwe          - JWE String
 * @param sharedSecret - Shared Secret
 * @return JWS in Compact Serialization format
 * @throws Exception
 */
public static String createJws(String jwe, String sharedSecret) throws
Exception {
    Map<String, Object> customParameters = new HashMap<>();
    long iat = System.currentTimeMillis() / 1000;
    Long exp = iat + 120;
    customParameters.put("iat", iat);
    customParameters.put("exp", exp);
    JWSHeader.Builder builder = new JWSHeader.Builder(JWSAlgorithm.HS256);
    builder.customParams(customParameters);
    JWSHeader jwsHeader = builder.build();
    JWSObject jwsObject = new JWSObject(jwsHeader, new Payload(jwe));
    jwsObject.sign(new
MACSigner(sharedSecret.getBytes(StandardCharsets.UTF_8)));
    return jwsObject.serialize();
}

/**
 * Verify And Extract JWE from JWS
 *
 * @param jws          - JWS in Compact Serialization format
 * @param sharedSecret - Shared Secret
 * @return JWE String
 * @throws Exception
 */
public static String verifyAndExtractJweFromJWS(String jws, String
sharedSecret) throws Exception {
    JWSObject jwsObject = JWSObject.parse(jws);
    if (!jwsObject.verify(new
MACVerifier(sharedSecret.getBytes(StandardCharsets.UTF_8)))) {
        throw new Exception("Invalid signature");
    }
    Map<String, Object> customParameters =
jwsObject.getHeader().getCustomParams();
    if (customParameters == null) {
        throw new Exception("Invalid signature");
    }
    Long now = System.currentTimeMillis() / 1000;
    if ((Long) customParameters.get("iat") > now || (Long)
customParameters.get("exp") < now) {
        throw new Exception("Invalid signature");
    }
    return jwsObject.getPayload().toString();
}
```

JWS Using RSA

The general approach for generating a JWS payload using RSA that signs a JWE payload is as follows:

- Visa uses the compact serialization style, wherein elements are separated by a period (“.”).
- All fields are Base64URL-encoded.
- Use RSA-OAEP-256 algorithm (PS256 in terms of JWS algorithm notation) for signature.

All string-to-byte and byte-to-string conversions are done with the UTF-8 charset.

JWS Header Sample for VISA Inbound APIs

```
{
        "kid": "49YOKJPH154IHNJ6L3CH13wElP5AZPInRG1zzQ6SjNHOPCw4s",
        "cty": "JWE",
        "typ": "JOSE",
        "alg": "PS256"
}
```

Note(s):

- The ***kid*** value is the signing certificate id which will get generated while uploading the client signing certificate in VDCS portal.
- Signing certificate will be provided by the client (Issuers) only. Ensure that Key identifier (KID) value is not any inbound or encryption APIkeys.
- Kid value is the signing certificate id which has to be provided by client and needs to be uploaded by them in VDCS portal. Please note that certificate usage is unused doesn't mean it's not used
- Ensure that Expiry date of the Signing certificate must be after the current date
- Make sure Issuer CN= and Subject CN= should be same. This ensures that it is a self-signed certificate.
- Algorithm used should be PS256
- Kid and alg values are mandatory in JWS header
- If user wants to view the certificate, download it from VDCS portal and save it in local computer in txt or pem format and use below command to view the certificate.
- Run the below command from the terminal and one can view the certificate
    - ***For example: openssl x509 -text -in <CERT>.txt***

JWS Header Sample for VISA Outbound APIs

The JWS creation for outbound calls is the reverse of the JWS creation of inbound calls.

```
{
        "kid": "715EA257 ",
        "cty": "JWE",
        "typ": "JOSE",
        "alg": "PS256"
}
```

Note: The ***kid*** value is the VISA Signing Certificate Id

General steps for creating a signature

To create a signature by applying JWS on JWE:

1. Get the RSA 2048 private key.
2. Sign the JWE (JWS payload) that is using the RSA 2048 private key by using the algorithm specified in the alg header field (PS256).
3. JWS Signing Input:

ASCII (Base64URL (UTF8 (JWS Header)) || '.' || Base64URL(JWS Payload))

4. Package the header by using alg,kid,cty and typ params (as shown in the above example for JWS).
5. Base64URL-encode the JWS signature to produce an E-Signature.
6. Base64URL-encode the JWE to produce E-JWS payload.
7. Base64URL-encode the entire JWS header JSON containing the encryption parameters.
8. Concatenate the above elements to produce the JWS:

Base64URL (UTF8 (JWS Header)) || '.' || Base64URL (JWE) || '.' || Base64URL (JWS Signature)

Signature Validation

General steps to validate a signature:

1. Get the RSA 2048 public Key.

   Below is the command for generating the public key from the Signing certificate.

   openssl> x509 -pubkey -noout -in certificate.pem > pubkey.pem

2. Decode the E-Header field.
3. Get the kid and the alg elements to use for signature validation.

4. Decode the E-Signature field to get the JWS signature.
5. Decode the E-JWS payload to get the JWS payload.
6. Validate the JWS signature against the JWS payload by using the RSA 2048 public key with the encryption algorithm, as specified in alg element (PS256).

Java Sample Code

```
import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.RSASSASigner;
import com.nimbusds.jose.crypto.RSASSAVerifier;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.X509EncodedKeySpec;

public class JWSUtilsUsingRSAPKI {
    /**
     * Create JWS Using RSA PKI
     *
     * @param jwe           - JWE Payload
     * @param rsaPrivateKey - RSA Private Key
     * @param signKid       - Signing Key UserID
     * @return JWS Compact Serialization Format
     * @throws JOSEException
     */
    public static String createJws(String jwe, PrivateKey rsaPrivateKey,
String signKid) throws JOSEException {
        JWSObject jwsObject = new JWSObject((new
com.nimbusds.jose.JWSHeader.Builder(JWSAlgorithm.PS256))

.type(JOSEObjectType.JOSE).keyID(signKid).contentType("JWE").build(), new
Payload(jwe));
        JWSSigner signer = new RSASSASigner(rsaPrivateKey);
        jwsObject.sign(signer);
        return jwsObject.serialize();
    }
    /**
     * Verify A JWS
     *
     * @param jws       - JWS Compact Serialization Format
     * @param publicKey - Public Key
     * @return
     */
    public static boolean verifyJws(String jws, byte[] publicKey) {
        try {
            RSAPublicKey rsaPublicKey = (RSAPublicKey)
KeyFactory.getInstance("RSA").generatePublic(new
                    X509EncodedKeySpec(publicKey));
            JWSVerifier verifier = new RSASSAVerifier(rsaPublicKey);
            JWSObject jwsObject = JWSObject.parse(jws);
            return jwsObject.verify(verifier);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

Appendix

List of JOSE algorithms

The following table lists the JOSE algorithm support of the Java 7, Java 8 and BouncyCastle JCA providers.

Algorithm family	Java 7	Java 8	Bouncy Castle
JWS algorithms			
HS256, HS384, HS512	✓	✓	✓
RS256, RS384, RS512	✓	✓	✓
PS256, PS384, PS512	✗	✗	✓
ES256, ES384, ES512, ES256K	✓	✓	✓
JWE algorithms			
RSA1_5	✓	✓	✓
RSA-OAEP, RSA-OAEP-256	✓	✓	✓
A128KW, A192KW, A256KW	✓	✓	✓
ECDH-ES, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW	✓	✓	✓
A128GCMKW, A192GCMKW, A256GCMKW	✗	✓	✓
PBES2-HS256+A128KW, PBES2-HS384+A192KW, PBES2-HS512+A256KW	✓	✓	✓
JWE algorithms			
A128CBC-HS256, A192CBC-HS384, A256CBC-HS512	✓	✓	✓
A128GCM, A192GCM, A256GCM	✗	✓	✓
A128CBC+HS256, A256CBC+HS512 (deprecated)	✓	✓	✓

JWS and JWE Headers list for Visa inbound and outbound calls for Issuers.

- The ***kid*** value is the signing certificate id which will get generated while uploading the client signing certificate in VDCS portal.
- Signing certificate will be provided by the client (Issuers) only. Ensure that Key identifier (KID) value is not any inbound or encryption API keys.
- Kid value is the signing certificate id which has to be provided by client and needs to be uploaded by them in VDCS portal. Please note that certificate usage is unused doesn't mean it's not used
- Ensure that Expiry date of the Signing certificate must be after the current date
- Make sure Issuer CN= and Subject CN= should be same. This ensures that it is a self-signed certificate.

JWS Visa inbound	JWS Visa outbound
JWS kid must be client signing cert id JWS (shared by Client) • KID: VISA will send the kid value once when the certificates are received and uploaded in VDCS. • Fingerprint (SHA-1): share with us. • Serial Number : share with us.	JWS kid must be Visa signing cert id JWS (shared by VISA) • KID: ***715EA257*** • Fingerprint (SHA1) E6:69:0F:17:C3:E0:4C:CC:87:BD:71:C8:F7:9E:E4:14:D2:DD:61:FC • Serial Number B3:62:3A:03:5F:8D:71:94:15:06:03:03:06:73:25:4D (238441832984649924529694087209246991693)
JWE kid must be Visa encryption cert id JWE (shared by VISA) • KID: **83F4EEB2** • Fingerprint (SHA-1) 70:D6:93:8E:FC:11:B5:EE:B8:A0:F3:1E:4C:E2:E3:AB:22:AF:7A:8A	JWE header kid must be client encryption cert id JWE (shared by Client) • KID: VISA will send when we receive certs. • Fingerprint (SHA-1) share with us • Serial Number share with us.

FAQs

What are the possible reasons for JWS verification failures?

- When Issuers pass an incorrect kid value (for example: Apikey, encryption certid)
- When Issuers use an unsupported library other than nimbus libraries
- When Issuers use other algorithms other than PS256.

What are the possible reasons for JWE encryption failures?

- When client (Issuer) uses an incorrect certificate (signing cert instead of visa encryption cert) while generating the JWS.
- When client (Issuer) uses an incorrect KID in JWE header to cause the Encryption failures.
- When client use mismatch alg value with different enc algorithms.

What are the recommended algorithms?

- JWS header should be PS256 algorithm.
- JWE header should be alg: RSA_OAEP_256 for cek and encryption algorithm is: A256GCM.

Command to compare the certificates

openssl x509 -noout -modulus -in <CERT>.txt | openssl md

The modulus value is expected to be the same.