JWE-JWS Developer Guide

Developer Guide for Using JSON Object Signing and Encryption (JOSE) - JSON Web Encryption (JWE) and JSON Web Signature (JWS)

Version 1.0.1

February 7, 2023

Visa Confidential

REVISION HISTORY

Version	Date	Comments	Created/Updated By
1.0.0	05-July-2021	Initial version	Purnachandra Motati (pmotati@visa.com) Shameem Peerbuccoss (speerbuc@visa.com)
1.0.1	10-Jan-2023	Document updates: • Reviewed document content • Added JWE headers for each product • Updated FAQs	Shameem Peerbuccoss (speerbuc@visa.com)

TABLE OF CONTENTS

INTRODUCTION

This document describes the mechanism which Visa uses to encrypt and decrypt sensitive data such as PAN, cardholder name, , address, etc.

Visa uses the JSON Object Signing and Encryption (JOSE) technologies which is an IETF standards that provides a general approach for encrypting and signing any content. The technologies include the JSON Web Encryption (JWE) for encryption and JSON Web Signature (((JWS))) for signing.

Field-Level Encryption (FLE) and Message Level Encryption (MLE) are used to encrypt sensitive data when making an API request to Visa.to Visa. API responses may also contain encrypted data, which client will have to decrypt.

Field-level encryption is the method of encrypting specific data field(s) whereas Message-level encryption encrypts the entire message.

In case of FLE, the field name prefix "enc" of the request or response payload indicate that the field’s data is encrypted.

The following Visa products (APIs), not limited to and where applicable, uses the JSON Web Encryption (JWE) and Signing (JWS) :

- Visa Direct (VD)
- Visa In-App Provisioning (VIAP)
- Visa Token Service (VTS)
- Visa Token Service Provisioning (VTIS) and Lifecycle Management (LCM)
- Visa Installment Solutions (VIS)
- Visa Acceptance Cloud (VAC)

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 and Shared Secret) and asymmetric algorithms (JWE using RSA PKI).

For symmetric keys, the corresponding key ID or API Key and shared secret are assigned to the client during onboarding. 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.

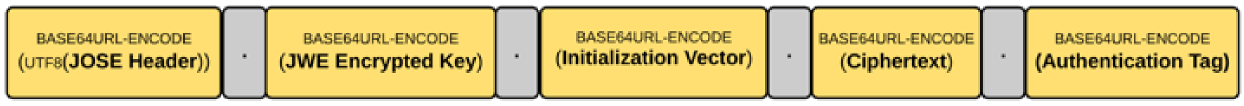
- Similarly, Visa will encrypt all necessary fields in the response payload using the client's public key and client will decrypt these fields using their private key.

Refer to the product-specific API specification document or the implementation guide for more information on which type of encryption is required for the Visa product you are integrating with.

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)


BASE64URL-ENCODE (UTF8(JOSE Header))
.
BASE64URL-ENCODE (JWE Encrypted Key)
.
BASE64URL-ENCODE (Initialization Vector)
.
BASE64URL-ENCODE (Ciphertext)
.
BASE64URL-ENCODE (Authentication Tag)

Base64 encoding & conventions that Visa follows:

Visa has chosen to suffix ".." to the Base64URL encoded values, instead of the more popular == convention.

JWE HEADER

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

Name	Description
alg	(*Required*) 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	(*Required*) 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	(*Optional*) 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	(*Required*) Type of encryption used by the CEK to encrypt sensitive payload elements. Format: Values are A128GCM and A256GCM.

iat	(*Optional*) 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: Epoch time in seconds
exp	(*Optional*) Expiration time of the payload. This field is optional and will be evaluated in conjunction with the ttl default defined with use case (or product). Ttl (time to live) is a value for the period of time (in seconds) the message is valid for. exp can be used to reduce the default ONLY. If exp represents a time greater than iat + ttl then iat + ttl will be used. exp is calculated as iat + ttl Format: Epoch time in seconds
iss	(*Optional*) 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.
aud	(*Optional*) List of desired recipients, which are the token requestors' vClientID values. This field is required for pushing card enrollment to 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. (*Optional*) Format: String; alphanumeric; max length 256 characters.
channelSecurityContext	This indicates which channel security to use for encryption & decryption Values: RSA_PKI, RSA_PKI2, OPACITY_A, OPACITY_B, SHARED_SECRET
jti	(*Optional*) The "jti" (JWT ID) claim provides a unique identifier for the JWT.

Visa requires a different set of header fields to be provided. The below table shows the JWE header to be used for each product.

You can refer to the corresponding API specification for additional information.

- Visa Direct (VD)

JWE Header	Value
alg	RSA-OAEP-256
typ	JOSE
kid	<MLE KEY ID>
enc	A128GCM

Iat	Time in UTC when the JWE was issued iat is not required for P2W (Push to Wallet) and P2A (Push to Account)

- Visa In-App Provisioning (VIAP)

JWE Header	Value
alg	RSA-OAEP-256
typ	JOSE
kid	< ENCRYPTION CERTIFICATE ID >
enc	A256GCM

- Visa Token Service (VTS)

JWE Header	Value
JWE Using API Key and Shared Secret	
alg	A256GCMKW
typ	JOSE
kid	<API KEY>
enc	A256GCM
iat	Time in UTC when the JWE was issued iat is mandatory for push provisioning. It is also conditionally required when the exp header is present.
exp	Expiration time of the payload. This field is optional and should be evaluated in conjunction with the ttl default.
iss	Message originator client id for VCEH (Visa Card Enrollment Hub) use case
aud	Client ID of desired recipient for VCEH (Visa Card Enrollment Hub) use case
JWE Using Opacity A	
alg	A256GCMKW
typ	JOSE
kid	<CLIENT DEVICE ID>
enc	A256GCM
channelSecurityContext	Opacity_A
JWE Using RSA PKI	
alg	RSA1_5
typ	JOSE
kid	<CLIENT DEVICE ID>
enc	A128GCM
channelSecurityContext	RSA_PKI
iat	Time in UTC when the JWE was issued
JWE Using RSA PKI2	
alg	RSA-OAEP-256
typ	JOSE
kid	<CLIENT DEVICE ID>
enc	A128GCM

channelSecurityContext	RSA_PKI2
iat	Time in UTC when the JWE was issued

- Visa Token Service Provisioning (VTIS) and Lifecycle Management (LCM)

JWE Header	Value
alg	RSA-OAEP-256
typ	JOSE
kid	**Sandbox**: 83F4EEB2 **Production**: 77F95D98
enc	A256GCM
iat	Time in UTC when the JWE was issued iat is mandatory for push provisioning. It is also conditionally required when the exp header is present.
exp	Expiration time of the payload. This field is optional and should be evaluated in conjunction with the ttl default.
iss	Message originator client id for VCEH (Visa Card Enrollment Hub) use case
aud	Client ID of desired recipient for VCEH (Visa Card Enrollment Hub) use case

- Visa Installment Solutions (VIS)

JWE Header	Value
alg	A256GCMKW
typ	JOSE
kid	<ENCRYPTION KEY ID>
enc	AGCM

- Visa Acceptance Cloud (VAC)

JWE Header	Value
alg	RSA-OAEP-256
typ	JOSE
kid	<ENCRYPTION CERTIFICATE ID>
enc	A256GCM

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: '<Encryption API Key>'
- enc: '<Encryption Algorithm to be used.>'
- iv: '<A unique 96-bit Base64 URL 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 must be separated by “.”).
- All fields are Base64 – 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.

- Get the shared secret for the Encryption API key.
- Create a SHA-256 digest of the shared secret.
- Generate a random Content Encryption Key (CEK)
- 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.
- Generate a random IV of length 96 bit. Base64-UrlSafe encode to produce Payload-IV.
- 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.
- Base64-UrlSafe encode the encrypted bytes to produce ciphertext.
- Base64-UrlSafe encode the Tag data to produce TAG.
- 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": "<Encryption API Key>", //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

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

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: '<***Visa Encryption Certificate Name***>
- 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:

- Get the RSA 2048 public key.
- Generate a Random Symmetric Key (RSK) of 256 bits length.
- Encrypt the RSK that is using the RSA 2048 key, by using the algorithm specified in alg (RSA-OAEP-256).
- Generate a random IV of 96 bits length and perform Base64URL encoding to produce E-IV.
- 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.
- Base64URL-encode the encrypted bytes to produce E-Ciphertext.
- Base64URL-encode the Tag data to produce E-TAG.
- Base64URL-encode the RSK to produce E-Encrypted Key.
- 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": "83F4EEB2",
  "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...RlcuYv1lwIvzw="

DECRYPTING A JWE USING RSA

- Get the RSA 2048 private key.
- Decode the Base64URL E-Header field.
- Get the kid and the algorithms to use for decryption.
- Decode the Base64URL E-Key field to get the encrypted E-key (Random Symmetric Key).
- 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).
- JWE header (the base64 encoded string in bytes) is used to pass AAD (Additional Authentication Data) for the A256GCM operation.
- Get the RSK in clear.
- Decode the Base64URL E-IV for use in ciphertext decryption.
- Decode the Base64URL E-Authentication Tag for use in ciphertext decryption.
- Decode the Base64URL ciphertext field.
- Decrypt the ciphertext, by using RSK and IV and authentication tag and encryption algorithm, as specified in the enc header field (A256GCM).

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

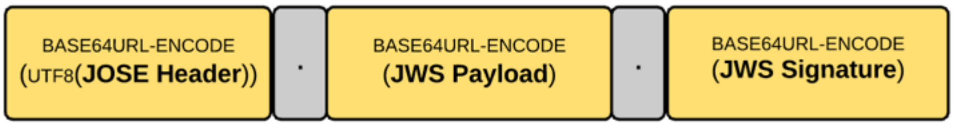
JWS represents signed content using JSON data structures and base64url encoding. The representation consists of three parts:

- 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 JWS header contains the metadata for signing.

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 string / content.
Iat	(Optional) 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:** Epoch time in seconds
exp	(Optional) Expiration time of the payload. This field is optional and will be evaluated in conjunction with the ttl default defined with use case (or product). Ttl (time to live) is a value for the period of time (in seconds) the message is valid for. exp can be used to reduce the default ONLY. If exp represents a time greater than iat + ttl then iat + ttl will be used. exp is calculated as iat + ttl **Format:** Epoch time in seconds

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 JWE payload.

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

Below is an example of the JWS Header 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 Visa System.
- Signing certificate will be provided by the client. Ensure that Key identifier (KID) value is not for an inbound or encryption certificate.
- Kid value is the signing certificate id which must be provided by client and needs to be uploaded by them in Visa System. 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 Visa System 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.

Below is an example of the JWS Header for Visa Outbound APIs:

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

Note: The ***kid*** value is the Visa Signing Certificate ID

GENERAL STEPS FOR CREATING A SIGNATURE (JWS)

To create a signature by applying JWS on JWE:

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

  ASCII (Base64URL (UTF8 (JWS Header)) || '.' || Base64URL(JWS Payload))
- Package the header by using alg, kid, cty and typ params (as shown in the above example for JWS).
- Base64URL-encode the JWS signature to produce an E-Signature.
- Base64URL-encode the JWE to produce E-JWS payload.
- Base64URL-encode the entire JWS header JSON containing the encryption parameters.
- Concatenate the above elements to produce the JWS:

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

SIGNATURE (JWS) VALIDATION

Below are the steps to validate a signature:

- 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

- Decode the E-Header field.
- Get the kid and the alg elements to use for signature validation.
- Decode the E-Signature field to get the JWS signature.
- Decode the E-JWS payload to get the JWS payload.
- 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).

APPENDIX

LIST OF JOSE ALGORITHMS

The following table lists the JOSE algorithm support of the Java 7, Java 8 and Bouncy Castle 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 TOKEN SERVICES ISSUER INBOUND AND OUTBOUND CALLS.

JWS Visa inbound (Client to Visa) - *This is incoming request to Visa from Client.*	**JWS Visa outbound (Visa to Client) -** *This is outgoing request from Visa to the Client such as notifications or Issuer APIs*
The JWS kid must be the Client Signing Certificate ID Client will create the signing certificate to the Visa Representative which will be uploaded in Visa System. Once uploaded, the KID value will be generated and share to the client Client will use the private key corresponding to the signing certificate to sign the payload to create the JWS	**The JWS kid will be the Visa Signing Certificate ID.** The Visa Signing Certificate or public key will be share with the client. The KID value will be ***715EA257*** and match the certificate's Subject DN
The JWE kid must be Visa Encryption Certificate ID. This ID will be shared to the client along with the Visa Encryption Certificate or public key. Kid Value: • **Sandbox: 83F4EEB2** • **Production: 77F95D98**	**The JWE header kid will be the Client Encryption Certificate ID** Client will create the encryption certificate or public key to the Visa Representative which will be uploaded in Visa System. Once uploaded, the KID value will be generated and share to the client. Visa will use the client encryption certificate or public key to encrypt the payload to create the JWE. Client will intern use the corresponding private key to decrypt the encrypted payload (JWE).

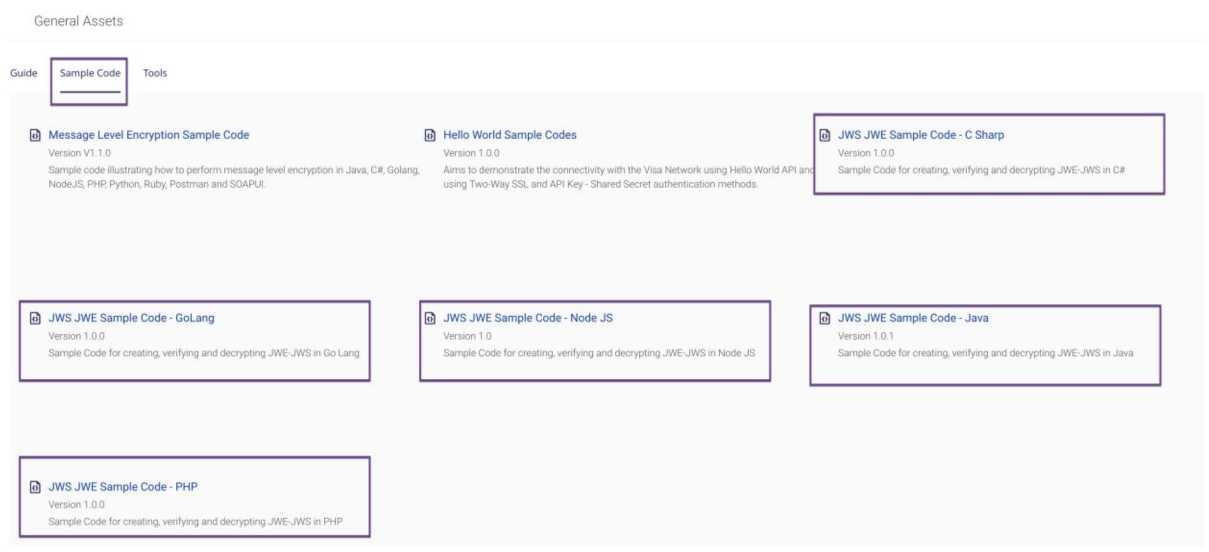
JWE JWS SAMPLE CODES ON VISA DEVELOPER PORTAL (VDP)

The JWE JWS sample codes are available on the Visa Developer Portal (VDP), under General Assets, which can be used as reference codes to speed up integration and development:

The sample codes are available in the following programming languages:

- C#
- Go Lang
- Nodes JS
- Java

- PHP


General Assets

Message Level Encryption Sample Code
Version V1.1.0
Sample code illustrating how to perform message level encryption in Java, C#, Golang, NodeJS, PHP, Python, Ruby, Postman and SOAPUI.

Hello World Sample Codes
Version 1.0.0
Aims to demonstrate the connectivity with the Visa Network using Hello World API and using Two-Way SSL and API Key - Shared Secret authentication methods.

JWS JWE Sample Code - C Sharp
Version 1.0.0
Sample Code for creating, verifying and decrypting JWE-JWS in C#

JWS JWE Sample Code - GoLang
Version 1.0.0
Sample Code for creating, verifying and decrypting JWE-JWS in Go Lang

JWS JWE Sample Code - Node JS
Version 1.0
Sample Code for creating, verifying and decrypting JWE-JWS in Node JS

JWS JWE Sample Code - Java
Version 1.0.1
Sample Code for creating, verifying and decrypting JWE-JWS in Java

JWS JWE Sample Code - PHP
Version 1.0.0
Sample Code for creating, verifying and decrypting JWE-JWS in PHP

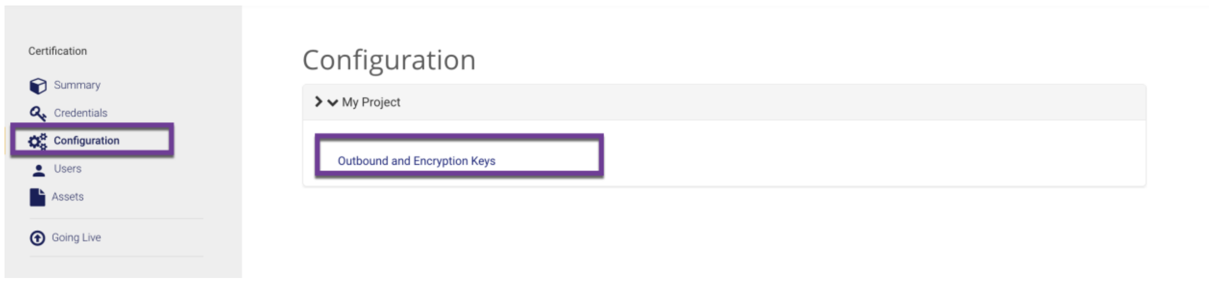
If you necessitate any sample code in other languages not available on the Visa Developer Portal (VDP), please reach out to your Visa Representative.

FAQS

WHERE CAN I FIND BUY PROJECT ENCRYPTION KEY AND SHARED SECRET FOR ENCRYPTION AND DECRYPTION?

An Encryption (API) key and shared secret can be obtained from your Visa Developer Portal (VDP) Project.

To view the project's Encryption Keys, click on the "Configuration" tab within the VDP project → Outbound and Encryption Keys.



If you don't have an API Key, you can create one by clicking on the "Create New" button.

WHERE CAN I FIND MY PROJECT ENCRYPTION KEY ID AND ENCRYPTION CERTIFICATE?

- If your project require MLE (Message Level Encryption), please see our VDP Article on How to run MLE Sample Code Project which explains on how to set up MLE and obtain your credentials.

- For VTS (Visa Tokenization Services), please contact your Visa Representative.

WHAT ARE THE POSSIBLE REASONS FOR JWE ENCRYPTION FAILURES?

Visa will respond with an HTTP 400 status code in the below reasons:

- Incorrect JWE string being sent in the request which Visa cannot parse and decrypt
- Incorrect Encryption kid or API Key value used in the JWE Header.
- Incorrect shared secret was used for Symmetric Encryption and decryption
- Incorrect certificate (signing cert instead of encryption cert) was used for Asymmetric Encryption.
- Incorrect private key used for signing the JWE or decrypting the JWE.
- Mismatch alg value with different enc algorithms in JWE.
- The JWE Header field such as Content Encrypted Key (CEK), Initialization Vector, Ciphertext and Authentication Tag were not Base64URL encoded.
- Incorrect values in the header fields like Initialization Vector or Authorization Tag.

WHAT ARE THE POSSIBLE REASONS FOR JWS VERIFICATION FAILURES?

Visa will respond with an HTTP 400 status code in the below reason:

- Incorrect JWS Payload being send in the request which Visa cannot parse or verify the signature
- Incorrect signing kid value in the JWS Header. For example: Encryption API Key or an Encryption Certificate ID was parsed instead of the Signing API Key or Signing Certificate ID
- Incorrect shared secret or different private key was used to sign the payload to create the JWS
- An unsupported library other than nimbus libraries was used
- Other algorithms were used.

WHAT ARE THE RECOMMENDED ALGORITHMS?

- JWE header should be:
  - **alg**: AGCM256KW for Symmetric encryption algorithm is: **A256GCM** for Shared Secret (See corresponding API Specifications)
  - alg: RSA_OAEP_256 for Asymmetric algorithm is: A256GCM for RSA PKI (See corresponding API Specifications)
- JWS header should be HS256 for Shared Secret (Symmetric) and PS256 algorithm for RSA PKI (Asymmetric). (See corresponding API Specifications)

HOW TO VERIFY IF YOUR CERTIFICATE OR PUBLIC KEY MATCH WITH YOUR PRIVATE KEY?

We will use the OpenSSL to compute the hash of the modulus of the certificate or public key and private key.

- Compute the modulus of the certificate or private key

openssl x509 -modulus -noout -in <CERT_OR PUBLIC_KEY_PEM_FILE> | openssl sha256

- Compute the modulus of the private key

openssl rsa -modulus -noout -in <PRIVATE_KEY_PEM_FILE> | openssl sha256

The computed SHA256 hash of the modulus in should be the same.