Connectivity & Security

1. Overview

This document outlines the connectivity and security mechanisms used by the Token Management Platform (TMP) for secure communication between external Issuer systems and TMP backend services. TMP uses X.509 mutual authentication and JWE (JSON Web Encryption) to ensure data confidentiality and endpoint authenticity.

2. Connectivity Architecture

2.1 Issuer ? TMP

- **Protocol:** HTTPS (TLS 1.2+)
- **Authentication:** Mutual TLS (X.509)
- **Issuer Role:** Client
- **TMP Role:** Server
- **Certificate Usage:**
  - The Issuer presents a client certificate signed by a Verestro CA.
  - TMP validates the Issuer's certificate against its trust store.
  - TMP authenticates the Issuer

2.2 TMP ? Issuer

- **Protocol:** HTTPS (TLS 1.2+)
- **Authentication:** Mutual TLS (X.509)
- **TMP Role:** Client
- **Issuer Role:** Server
- **Certificate Usage:**
  - TMP presents its client certificate.
  - The Issuer validates TMP’s certificate against its trust store.
  - The Issuer authenticates TMP

2.3 Certificate Generation and Use

Certificate Type	Generated By	Signed By	Used By	Purpose(s)
Client Certificate (Issuer)	Issuer	Verestro	Issuer	X.509 mTLS authentication (Issuer → Verestro)

Certificate Type	Generated By	Signed By	UsedBy	Purpose(s)
			Verestro/Issuer	JWE. Verestro uses the public key from this certificate to encrypt sensitive fields (e.g., card number) in requests. The Issuer uses the private key from this certificate to decrypt sensitive fields
Client Certificate (Verestro)	Verestro	Issuer	Verestro	X.509 mTLS authentication (Verestro → Issuer)
			Issuer/ Verestro	JWE. The Issuer uses the public key from this certificate to encrypt sensitive fields (e.g., card number) in requests. Verestro uses the private key from this certificate to decrypt sensitive fields

> **Note:** While the same certificate pair may be used for both mTLS and JWE, it's **recommended** to use **separate certificates** for TLS and field-level encryption for improved security and certificate lifecycle management.

3. Certificate Requirements

- **Format:** X.509
- **Key Length:** RSA 2048-bit or higher
- **Validity:** Minimum 1 year validity recommended

Certificates are exchanged securely during partner onboarding. The same certificate can be used for both mutual TLS and field-level encryption (JWE).

Check the following document on how to create a client certificate:

https://developer.verestro.com/books/connecting-to-our-services-and-sandbox/page/connecting-to-server-to-server-apis-fe-sandbox

4. Field-Level Encryption

Certain sensitive fields (e.g., `cardNumber`) are encrypted using JWE (JSON Web Encryption) to provide end-to-end protection beyond TLS.

4.1 JWE Encryption Details

Data is encrypted using JWE per RFC 7516 (https://tools.ietf.org/html/rfc7516)

JWE header	Name	Description

alg	RSA-OAEP-256	Cryptographic algorithm used to encrypt CEK
enc	A256GCM	Identifies the content encryption algorithm used to perform authenticated encryption

- **Algorithm:** RSA-OAEP-256
- **Content Encryption:** A256GCM
- **Key Material:** Derived from or aligned with the public key in the X.509 certificate
- **Envelope Format:** Compact JWE serialization

4.2 Encrypted Fields

Field	Location	Requirement
`encryptedCardNumber`	JSON Payload	Must be encrypted
`encryptedCVC`	JSON Payload	Must be encrypted if present

The same certificate used for TLS client authentication may be used for encrypting JWE payloads, leveraging the associated public key.

4.3 Example code

Check the example code

Encryption

```
import org.apache.commons.codec.digest.DigestUtils;
import com.nimbusds.jose.*;
import com.nimbusds.jose.crypto.RSAEncrypter;

public String encryptCardNumberExample() throws Exception {
 final String cardNumber = "1234123412341234"; //plain text card number
 final String publicKey = "-----BEGIN PUBLIC KEY-----*****-----END PUBLIC KEY-----"; //
Verestro public key

 final String publicKeyContent = publicKey.replaceAll("\\n", "")
                                          .replace("-----BEGIN PUBLIC KEY-----", "")
                                          .replace("-----END PUBLIC KEY-----", "");
 KeyFactory factory = KeyFactory.getInstance("RSA");
 X509EncodedKeySpec encodedKeySpec = new
X509EncodedKeySpec(Base64.getDecoder().decode(publicKeyContent));
```

```
 final RSAPublicKey rsaPublicKey = (RSAPublicKey) factory.generatePublic(encodedKeySpec);

 JWEHeader.Builder headerBuilder = new JWEHeader.Builder(JWEAlgorithm.RSA_OAEP_256,
EncryptionMethod.A256GCM);
 JWEHeader header = headerBuilder.type(JOSEObjectType.JOSE)
                                 .customParam("iat", Instant.now().getEpochSecond()) //
issued at timestamp
 .keyID(DigestUtils.sha1Hex(rsaPublicKey.getEncoded()))
                                 .build();
 JWEObject object = new JWEObject(header, new Payload(cardNumber.getBytes()));

 object.encrypt(new RSAEncrypter(rsaPublicKey));
 final String jweEncryptedCardNumber = object.serialize(); // encrypted card number string
 return jweEncryptedCardNumber;
}
```

Decryption

```
import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.JWEObject;
import com.nimbusds.jose.crypto.RSADecrypter;
import org.springframework.core.io.DefaultResourceLoader;
import org.springframework.core.io.Resource;

import java.io.IOException;
import java.io.InputStream;
import java.security.*;
import java.security.cert.CertificateException;
import java.text.ParseException;

import static java.util.Objects.*;

public class JweDecryptor {

    public String decrypt(String encryptedString) {
        try {
           JWEObject jweObject = JWEObject.parse(encryptedString);
           KeyStore keyStore = loadKeystore("/keystore.p12", "password");
           PrivateKey privateKey = getPrivateKey(keyStore);
```

```
            jweObject.decrypt(new RSADecrypter(privateKey));
            return new String(jweObject.getPayload().toBytes());
        } catch (ParseException | JOSEException e) {
                    throw new IllegalStateException("Error on decryption JWE payload");
        }
    }

    private KeyStore loadKeystore(String storeLocation, String storePassword) {
         Resource resource = new DefaultResourceLoader().getResource(storeLocation);
        try (InputStream inputStream = resource.getInputStream()) {
                    requireNonNull(inputStream, "Resource could not be loaded" +
storeLocation);
        KeyStore certificateResource = KeyStore.getInstance(KeyStore.getDefaultType());
        certificateResource.load(inputStream, storePassword.toCharArray());
        return certificateResource;
        } catch (CertificateException | KeyStoreException | IOException |
NoSuchAlgorithmException e) {
                    throw new IllegalStateException(e);
        }
    }

    private PrivateKey getPrivateKey(KeyStore keyStore) {
          try {
            return (PrivateKey) keyStore.getKey("alias", "password".toCharArray());
          } catch (KeyStoreException | NoSuchAlgorithmException | UnrecoverableKeyException
e) {
                      throw new IllegalStateException(e);
          }
    }
}
```

Revision #7
Created 12 May 2025 13:20:49
Updated 13 May 2025 08:02:04