Signing and Validating MATF Metadata

This guide outlines the steps to implement MATF metadata.

Contents

Prerequisites

Prior to starting, ensure you have a working understanding of MATF, JSON Web Signature (JWS), and JSON Schema.

- Familiarize yourself with the MATF draft specification, available at: MATF Draft Specification
- Refer to the JWS specification found at: JWS Specification
- Explore JSON Schema through the official website: JSON Schema

Base64 vs. Base64url Encoding

Before proceeding, it's crucial to understand the difference between Base64 and Base64url encoding. The MATF metadata uses Base64url encoding for various components, including the JWS header, payload, and signature. The key distinction is that Base64url encoding is URL-safe and does not include characters that are problematic in URLs, such as '+' and '/' in Base64. Instead, it uses '-' and '_' to replace these characters. Be sure to use Base64url encoding when working with JWS components.

Implementation Steps

Create a Signing Key

To begin, you'll need to create a signing key and certificate that will be used when signing the JWS. Generate a self-signed certificate for this purpose. Self-signed certificates offer the advantage of flexibility in setting their expiration period, which can be chosen for a more extended duration, often measured in years.

The self-signed certificate's primary function, when validating the JWS signature, is to provide the public key. It's important to note that other attributes of the certificate, such as the Common Name (CN) and Subject Alternative Names (SANs), will not be used or validated during the JWS signature verification process. This underscores why obtaining a certificate from a Certificate Authority (CA) with extensive CN and SAN validations is unnecessary for this specific use case. A self-signed certificate suffices for providing the required public key, streamlining the process without the need for additional, CA-validated attributes. Moreover, the flexibility to set a longer expiration time for self-signed certificates reduces the administrative burden of frequent certificate renewals while ensuring the security of the JWS signature.

Here are the commands to create the signing key and certificate using OpenSSL:

Generate the EC private key:

```
openssl ecparam -genkey -name prime256v1 -noout -out ec-key.pem
```

Create a self-signed certificate with a long validity period:

```
openssl req -new -x509 -key ec-key.pem -out cert.pem -outform pem -days 36500 -subj '/CN=Signer'
```

Create the JSON Web Signature (JWS)

Create the JSON Web Signature (JWS) following the specified format and content, using the general JWS JSON Serialization syntax. Make sure to include the necessary claims, such as iss, exp, iat, version, cache_ttl, and entities, as outlined in the specification.

Understanding the `cache_ttl` Claim

The cache_ttl claim is a critical component of the JWS. It defines the time-to-live duration in seconds for caching the data within the JWS. This duration governs how long the metadata can be cached before it needs to be refreshed by fetching updated data.

It is essential to understand the importance of the cache_ttl claim. It ensures that the information contained within the JWS remains fresh and accurate. Therefore, the metadata should be retrieved and updated within this specified cache TTL period.

Understanding the exp Claim

In addition to the cache_ttl claim, it's vital to understand the significance of the exp claim, short for "Expiration Time". The exp claim specifies the timestamp after which the data contained within the JWS is no longer considered valid. Beyond this timestamp, the data should not be considered reliable or usable.

The exp claim is particularly important when it's not possible to fetch updated data within the cache_ttl period. It safeguards against using outdated or potentially inaccurate information.

The following is a non-normative example of a metadata statement. Line breaks within the issuers' claim are for readability only.

```
{
  "version": "1.0.0",
  "cache_ttl": 3600,
  "entities": [{
    "entity_id": "https://example.com",
    "organization": "Example Org",
    "issuers": [{
      "x509certificate": "-----BEGIN CERTIFICATE-----\nMIIDDDCCAf
      SgAwIBAgIJAIOsfJBStJQhMA0GCSqGSIb3DQEBCwUAMBsxGTAXBgNV\nBAM
      MEHNjaW0uZXhhbXBsZS5jb20wHhcNMTcwNDA2MDc1MzE3WhcNMTcwNTA2MD
      c1\nMzE3WjAbMRkwFwYDVQQDDBBzY2ltLmV4YW1wbGUuY29tMIIBIjANBgk
      qhkiG9w0B\nAQEFAAOCAQ8AMIIBCgKCAQEAyr+3dXTC8YXoi0LDJTH0lTfv
      8omQivWFOr3+/PBE\n6hmpLSNXK/EZJBD6ZT4Q+tY8dPhyhzT5RFZCVlrDs
      e/kY00F4yoflKiqx9WSuCrq\nZFr1AUtIfGR/LvRUvDFtuHo1MzFttiK8Wr
      wskMYZrw1zLHTIVwBkfMw1qr2XzxFK\njt0CcDmFxNdY5Q8kuBojH9+xt5s
      ZbrJ9AVH/OI8JamSqDjk9ODyGg+GrEZFClP/B\nxa4Fsl04En/9GfaJnCU1
      NpU0cqvWbVUlLOy8DaQMN14HIdkTdmegEsg2LR/XrJkt\nho16diAXrgS25
      3xbkdD3T5d6lHiZCL6UxkBh4ZHRcoftSwIDAQABo1MwUTAdBgNV\nHQ4EFg
      QUs1dXuhGhGc2UNb7ikn3t6cBuU34wHwYDVR0jBBgwFoAUs1dXuhGhGc2U\
      nNb7ikn3t6cBuU34wDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG9w0BAQsFAA
      OCAQEA\nrR9wxPhUa2XfQ0agAC0oC8TFf8wbTYb0ElP5Ej834xMMW/wWTSA
      N8/3WqOWNQJ23\nf0vEeYQwfvbD2fjLvYTyM2tSPOWrtQpKuvulIrxV7Zz8
      A61NIjblE3rfea1eC8my\nTkDOlMKV+wlXXgUxirride+6ubOWRGf92fgze
      DGJWkmm/a9tj0L/3e0xIXeujxC7\nMIt3p99teHjvnZQ7FiIBlvGc1o8FD1
      FKmFYd74s7RxrAusBEAAmBo3xyB89cFU0d\nKB2fkH2lkqiqkyOtjrlHPoy
      6ws6g1S6U/Jx9n0NEeEqCfzXnh9jEpxisSO+fBZER\npCwj2LMNPQxZBqBF
      oxbFPw==\n-----END CERTIFICATE-----"
    }],
    "servers": [{
      "description": "SCIM Server 1",
      "base_uri": "https://scim.example.com/",
      "pins": [{
        "alg": "sha256",
        "digest": "+hcmCjJEtLq4BRPhrILyhgn98Lhy6DaWdpmsBAgOLCQ="
      }],
      "tags": [
        "scim"
      ]
    }],
    "clients": [{
      "description": "SCIM Client 1",
      "pins": [{
        "alg": "sha256",
        "digest": "+hcmCjJEtLq4BRPhrILyhgn98Lhy6DaWdpmsBAgOLCQ="
      }]
    }]
  }]
}
```


Sign the Metadata with JWS

Sign the JWS using the recommended algorithm, ECDSA with P-256 and SHA-256 ("ES256"). Ensure that you include the required headers in the JWS, such as alg and iss, as specified in the specification.

Example JSON Web Signature (JWS):

Below is an example JSON Web Signature (JWS) structure. This JWS serves as a practical reference for understanding how to format and structure the metadata while incorporating the necessary claims and encoding.

The JWS consists of the following components:

- **Payload**: This section contains the metadata represented as a JSON object. It includes essential claims such as entity IDs, organization details, version information, and timestamps.
- **Signatures**: A list of JWS signatures, each consisting of a signature value and protected headers. These signatures are applied to the metadata to ensure its integrity and authenticity during transmission.
- **Protected Headers**: These headers are safeguarded and encompass details regarding the signing algorithm as well as other claims.

Here's an example.

JWS


```
{
        "payload":
"eyJjYWNoZV90dGwiOiAzNjAwLCAiZW50aXRpZXMiOiBbeyJjb25zdGl0dWVudHMiOiBbeyJvcmdhbml6YXRpb25faWQiOiAiU0UxMTIyMzM0NDU
1IiwgIm9yZ2FuaXphdGlvbl9uYW1lIjogIkV4YW1wbGUgT3JnIE9uZSJ9LCB7Im9yZ2FuaXphdGlvbl9pZCI6ICJTRTY2Nzc4ODk5MDAiLCAib3J
nYW5pemF0aW9uX25hbWUiOiAiRXhhbXBsZSBPcmcgVHdvIn1dLCAiZW50aXR5X2lkIjogImh0dHBzOi8vaWRwMS5leGFtcGxlLmNvbSJ9XSwgImV
4cCI6IDIwMTA1NTMwMTUsICJpYXQiOiAxNjk1MTkzMDE1LCAiaXNzIjogImh0dHBzOi8vbXktZG9tYWluLmV4YW1wbGUuY29tIiwgInZlcnNpb24
iOiAiMS4wLjAifQ",
        "signatures": [{
                "signature": "vz8VPL0oJG6tw663kNEgCcZJeJ1buEbLaQLFEsb128I-
hUb3EiRVXb691L4rk6URUvRLze1cR6myB9HieC_kow",
                "protected":
"eyJhbGciOiJFUzI1NiIsIng1dCNTMjU2IjoiZjdkMkV1eHMxVUdoMHA0LTdMZ2E1c05XTmh4MjVmeERyMldQODdib3ZKVSJ9"
        }]
}
```


Content-Type: application/jose+json

When transmitting or storing JWS objects, use the media type "application/jose+json" in your HTTP headers or content-type declarations to indicate that the content follows the JSON Web Signature (JWS) and JSON Object Signing and Encryption (JOSE) standards.

Publish the Metadata

Share the JWS at a URL accessible to entities within the federation.

Validate and Interpret Metadata

Entities within the federation can retrieve and validate the metadata by accessing the URL specified by the federation. After receiving the JWS-signed metadata, it's crucial to perform validation and interpretation to ensure the integrity and authenticity of the data.

Verify the Digital Signature

1. **Validate the Digital Signature:** Execute the signature validation process, which involves fetching the JWS signature and confirming its validity using the federation's public key. Ensure alignment with the algorithm specified in the header **alg** (Algorithm) claim.
2. **Check the exp Claim**: Verify the **exp** (Expiration Time) claim in the JWS payload. Ensure that the current timestamp is before the specified expiration time. If the data is past its expiration time, it should not be considered valid.
3. **Check the Issuer (iss) Claim**: Verify that the **iss** (Issuer) claim in the JWS payload matches the expected issuer URI. This ensures that the metadata is coming from a trusted source.
4. **Validate the iat Claim**: Ensure that the **iat** (Issued At) claim is a valid NumericDate representing the time when the data was issued.

Validate Against JSON Schema

To ensure the metadata's structure and data types conform to the specification, you should validate it against the provided JSON Schema. The JSON Schema defines the expected format and structure of the metadata.

- Retrieve the JSON Schema: https://www.fedtls.se/schema
- Validate the received metadata against the JSON Schema to ensure it adheres to the defined structure and data constraints.

Interpret the Metadata

Once the metadata is validated:

- Extract and use the information as needed.
- Pay special attention to the cache_ttl and exp claims. The cache_ttl specifies the cache duration, while exp defines the expiration time of the data. Refresh the metadata before the cache_ttl expires or if exp is reached to maintain data accuracy.

By following these validation steps and interpreting the metadata correctly, you can trust the information provided within the JWS-signed metadata, enhancing the security and reliability of the federation.

Example Code for JWS Creation and Validation (TBD)

To facilitate the creation and validation of MATF metadata, you can leverage the example code available on GitHub. This code provides a practical reference for implementing the JWS generation and validation process.

Example Code Repository: (insert repo)

The provided example code offers a step-by-step guide, including the following functionalities:

- **Creating JWS**: The code demonstrates how to construct a JWS according to the specification's requirements, including the necessary claims.
- **Signing JWS**: It shows how to sign the JWS using the recommended algorithm (ECDSA with P-256 and SHA-256, "ES256") and include the required headers.
- **Serializing JWS:** The code illustrates how to serialize the JWS using the general JWS JSON Serialization syntax, ensuring Base64url encoding for various components.
- **Validating JWS**: It demonstrates how to validate the JWS signature, ensuring the integrity and authenticity of the metadata.
- **Validate Payload**: the code utilizes the MATF JSON schema to validate the data transmitted in the payload.

By utilizing this example code as a reference, you can streamline the implementation of MATF metadata. It provides practical guidance on handling JWS creation and validation, helping you ensure compliance with the specification's encoding and security requirements.

Tools

Convert PEM Certificate to JSON String Using Sed

```
sed ':a;N;$!ba;s/\n/\\n/g' <certificate.pem>
```

Create a pin with OpenSSL

```
openssl x509 -in <certificate.pem> -pubkey -noout | \
openssl pkey -pubin -outform der | \
openssl dgst -sha256 -binary | \
openssl enc -base64
```