eIDAS-Node and SAML 2.6

eIDAS eID Implementation

Exported on 04/15/2022

Table of Contents

Document history

Version	Date	Modification reason	Modified by
1.0	06/10/2017	Origination	DIGIT
2.0	11/04/2018	Editorial improvements	DIGIT
2.1	06/07/2018	Reuse of document policy updated and version changed to match the corresponding Release.	DIGIT
2.4	06/12/2019	Update regarding key agreement implementation, Other minor changes.	DIGIT
2.5	11/12/2020	eIDAS-Node 2.5 release	DIGIT
2.6	04/2022	eIDAS-Node 2.6 release	DIGIT

Disclaimer
This document is for informational purposes only and the Commission cannot be held responsible for any use which may be made of the information contained therein. References to legal acts or documentation of the European Union (EU) cannot be perceived as amending legislation in force or other EU documentation.
The document contains a brief overview of technical nature and is not supplementing or amending terms and conditions of any procurement procedure; therefore, no compensation claim can be based on the contents of the present document.

© European Union, 2022
Reuse of this document is authorised provided the source is acknowledged. The Commission's reuse policy is implemented by Commission Decision 2011/833/EU of 12 December 2011 on the reuse of Commission documents.

1 Introduction

This document is intended for a technical audience consisting of developers, administrators and those requiring detailed technical information on how to configure, build and deploy the eIDAS-Node application.
This document describes the W3C recommendations and how SAML XML encryption is implemented and integrated in eID.
Encryption of the sensitive data carried in SAML 2.0 Requests and Assertions is discussed alongside the use of AEAD algorithms as essential building blocks.

1.1 Document aims

The aim of this document is to describe how the eIDAS-Node implements SAML.

1.2 Document structure

This document is divided into the following sections:

- Chapter 1 – *Introduction* this section.
- Chapter 2 – *SAML Overview* provides an overview of.
- Chapter 3 – *eIDAS-Node and SAML XML encryption* provides information on the encryption of SAML messages.
- Chapter 4 – *eIDAS Node and SAML Metadata* describes the role of metadata in an eID scheme.
- Chapter 5 – *eIDAS-Node Protocol Engine* describes how the Protocol Engine is implemented.
- Chapter 6 – *ProtocolEngine Configuration* describes how the Protocol Engine is configured.
- Chapter 7 – *References* contains a list of reference documents for further information.

1.3 Other technical reference documentation

We recommend that you also familiarise yourself with the following eID technical reference documents which are available on **Digital Home > eID**

- *eIDAS-Node Installation, Configuration and Integration Quick Start Guide* describes how to quickly install a Service Provider, eIDAS-Node Connector, eIDAS-Node Proxy Service and IdP from the distributions in the release package. The distributions provide preconfigured eIDAS-Node modules for running on each of the supported application servers.
- *eIDAS-Node Installation and Configuration Guide* describes the steps involved when implementing a Basic Setup and goes on to provide detailed information required for customisation and deployment.
- *eIDAS-Node National IdP and SP Integration Guide* provides guidance by recommending one way in which eID can be integrated into your national eID infrastructure.
- *eIDAS-Node Demo Tools Installation and Configuration Guide* describes the installation and configuration settings for Demo Tools (SP and IdP) supplied with the package for basic testing.
- *eIDAS-Node Error and Event Logging* provides information on the eID implementation of error and event logging as a building block for generating an audit trail of activity on the eIDAS Network. It describes the files that are generated, the file format, the components that are monitored and the events that are recorded.
- *eIDAS-Node Security Considerations* describes the security considerations that should be taken into account when implementing and operating your eIDAS-Node scheme.
- *eIDAS-Node Error Codes* contains tables showing the error codes that could be generated by components along with a description of the error, specific behaviour and, where relevant, possible operator actions to remedy the error.

2 SAML Overview

The following overview is reproduced courtesy of OASIS (Copyright © OASIS Open 2008) (see section 7 — *References*).

The OASIS Security Assertion Markup Language (SAML) standard defines an XML-based framework for describing and exchanging security information between on-line business partners. This security information is expressed in the form of portable SAML assertions that applications working across security domain boundaries can trust. The OASIS SAML standard defines precise syntax and rules for requesting, creating, communicating, and using these SAML assertions.

The OASIS Security Services Technical Committee (SSTC) develops and maintains the SAML standard.

2.1 Drivers of SAML Adoption

Why is SAML needed for exchanging security information? There are several drivers behind the adoption of the SAML standard, including:

- **Single Sign-On:** Over the years, various products have been marketed with the claim of providing support for web-based SSO. These products have typically relied on browser cookies to maintain user authentication state information so that re-authentication is not required each time the web user accesses the system. However, since browser cookies are never transmitted between DNS domains, the authentication state information in the cookies from one domain is never available to another domain. Therefore, these products have typically supported multi-domain SSO (MDSSO) through the use of proprietary mechanisms to pass the authentication state information between the domains. While the use of a single vendor's product may sometimes be viable within a single enterprise, business partners usually have heterogeneous environments that make the use of proprietary protocols impractical for MDSSO. SAML solves the MDSSO problem by providing a standard vendor-independent grammar and protocol for transferring information about a user from one web server to another independent of the server DNS domains.
- **Federated identity:** When online services wish to establish a collaborative application environment for their mutual users, not only must the systems be able to understand the protocol syntax and semantics involved in the exchange of information; they must also have a common understanding of who the user is that is referred to in the exchange. Users often have individual local user identities within the security domains of each partner with which they interact. Identity federation provides a means for these partner services to agree on and establish a common, shared name identifier to refer to the user in order to share information about the user across the organizational boundaries. The user is said to have a federated identity when partners have established such an agreement on how to refer to the user. From an administrative perspective, this type of sharing can help reduce identity management costs as multiple services do not need to independently collect and maintain identity-related data (e.g. passwords, identity attributes). In addition, administrators of these services usually do not have to manually establish and maintain the shared identifiers; rather control for this can reside with the user.
- **Web services and other industry standards:** SAML allows for its security assertion format to be used outside of a "native" SAML-based protocol context. This modularity has proved useful to other industry efforts addressing authorization services (IETF, OASIS), identity frameworks, web services (OASIS, Liberty Alliance), etc. The OASIS WS-Security Technical Committee has defined a profile for how to use SAML's rich assertion constructs within a WS-Security security token that can be used, for example, to secure web service SOAP message exchanges. In particular, the advantage offered by the use of a SAML assertion is that it provides a standards-based approach to the exchange of information, including attributes, that are not easily conveyed using other WS-Security token formats.

3 eIDAS-Node and SAML XML encryption

3.1 Requirement description

The primary requirement of encryption is to protect the citizen's personal data against malicious attacks, which could:

- disrupt operation;
- gather sensitive information from a citizen; or
- gain access to a citizen's environment.

As there is no control of the environment where the citizen operates, this environment cannot be trusted and additional security measures must be taken. For these security measures, only strong, standard algorithms and strong keys are used in line with industry standards. It is important to ensure that effective key management is in place.

For transport confidentiality (eIDAS-Node-to-eIDAS-Node) the following options were proposed:

- End-to-end SAML encryption.
  Please note that this is NOT end-to-end confidentiality between the IdP and the SP, but only between eIDAS-Nodes.
- Encryption at the application level should not be imposed; nothing should be imposed on the SP or IdP by eIDAS and related specifications, as those two components are under MS' authority, therefore out-of-scope of eIDAS.
- Data privacy and confidentiality focusing on the scenario where the personal information data is compromised while in the clear (i.e. unencrypted) in the user's browser.
- The encryption functionality should be easily configurable at the eIDAS-Node level.
  The encryption is enabled or disabled by configuration from an external source (file) to provide the possibility of switching it without rebuilding the application.

3.2 XML 1.1 Encryption Recommendation

The W3C defines a recommendation for XML encryption in XML Encryption Syntax and Processing[1] in which it "*specifies a process for encrypting data and representing the result in XML*".
The current version of the standard is 1.1.

3.3 Encryption

The following sections show types of encryption supported by eID:

- Encryption of an entire element;
- Symmetric key encryption.

3.3.1 Encryption of an entire element

In this example, the entire *CreditCard* element is encrypted from its start to end tags. The cardholder's *Name* is not considered sensitive and so remains 'in the clear' (i.e. unencrypted).

1 http://www.w3.org/TR/xmlenc-core1/

```
< PaymentInfo  xmlns = "…" >
    < Name >John Smith</ Name >
    < CreditCard  Limit = "5,000"
  Currency = "EUR" >
        < Number >4019 2445 0277
5567</ Number >
        < Issuer >Example Bank</
Issuer >
        < Expiration >04/02</
Expiration >
    </ CreditCard >
</ PaymentInfo >
```

```
< PaymentInfo  xmlns = "…" >
    < Name >John Smith</ Name >
    < EncryptedData  Type = "…"
  xmlns = "…" >
        < CipherData >
            < CipherValue >A23B45
C56</ CipherValue >
        </ CipherData >
    </ EncryptedData >
</ PaymentInfo >
```

Figure 1: Encryption of an entire element

3.3.2 Symmetric key encryption with key transport mechanism

When using key transport mechanism, shared secret key encryption algorithms are specified for encrypting and decrypting sensitive content.
They appear as *Algorithm* attribute values (line 9) to *EncryptionMethod* elements (line 8).
They are children of *EncryptedKey* (line 6) which is in turn a child of *ds:KeyInfo* (line 5) which is in turn a child of *EncryptedData* (line 2) (or another *EncryptedKey*).

```
1  < saml2 :EncryptedAssertion>
2   < xenc :EncryptedData  Id = "_6a47298ff6092f82d9e540479c46bdfb "
   Type = "http://www.w3.org/2001/04/xmlenc#Element" >
3    < xenc :EncryptionMethod  Algorithm = "http://www.w3.org/2009/
   xmlenc11#aes256-gcm"  />
4
5    < ds :KeyInfo>
6     < xenc :EncryptedKey  Id = "_74229d1928a63480c0895c79497d20fe" >
7
8      < xenc :EncryptionMethod
9          Algorithm = "http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"
10      < ds :DigestMethod  Algorithm = "http://www.w3.org/2001/04/
   xmlenc#sha256"  />
11     </ xenc :EncryptionMethod>
12
13     < ds :KeyInfo>
14      < ds :X509Data>
15       < ds :X509Certificate>MIIDJzC...8mYfX8/jw==</ ds :X509Certificat
   e>
16      </ ds :X509Data>
```

```
17      </ ds :KeyInfo>
18
19      < xenc :CipherData>
20       < xenc :CipherValue>bSYaL0cXyC...UjDBQ==</ xenc :CipherValue>
21      </ xenc :CipherData>
22
23     </ xenc :EncryptedKey>
24    </ ds :KeyInfo>
25
26    < xenc :CipherData>
27     < xenc :CipherValue>pTvKqOqq...kfBb1rw=</ xenc :CipherValue>
28    </ xenc :CipherData>
29   </ xenc :EncryptedData>
30 </ saml2 :EncryptedAssertion>
```

For further information, refer to http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/Overview.html#sec-Usage.

The following steps show the process when encrypting with one-time-use symmetric key.

1. Start XML encryption.
2. Retrieve recipient's existing Asymmetric Public key.
3. Create new Symmetric key.
4. Encrypt XML with Symmetric key.
5. Encrypt Symmetric key with RSA using Public key.
6. Include encrypted Symmetric key into XML with *EncryptedKey* element.
7. Send XML.

3.3.3 Symmetric key encryption with key agreement mechanism

Key Agreement encryption algorithms are specified for encrypting and decrypting sensitive content.

Key Agreement encryption differs from the key transport encryption by its definition in the *EncryptedKey* (line 6) where we can find the *AgreementMethod* (line 12) that is part of the *KeyInfo* (line 11).

The *AgreementMethod* (line 12) is made of 3 parts: a *KeyDerivationMethod*, an *OriginatorKeyInfo* and a *RecipientKeyInfo*.

```
1  < saml2 :EncryptedAssertion>
2   < xenc :EncryptedData  Id = "_6a47298ff6092f82d9e540479c46bdfb "
   Type = "http://www.w3.org/2001/04/xmlenc#Element" >
3    < xenc :EncryptionMethod  Algorithm = "http://www.w3.org/2009/
   xmlenc11#aes256-gcm"  />
4
5    < ds :KeyInfo>
6     < xenc :EncryptedKey  Id = "_74229d1928a63480c0895c79497d20fe" >
7
8      < xenc :EncryptionMethod
9          Algorithm = "http://www.w3.org/2001/04/xmlenc#kw-aes256"  />
```

```
10
11      < ds :KeyInfo>
12       < xenc :AgreementMethod  Algorithm = "http://www.w3.org/2009/
        xmlenc11#ECDH-EC" >
13        < xenc11 :KeyDerivationMethod  Algorithm = "http://www.w3.org/
        2009/xmlenc11#ConcatKDF" >
14         < xenc11 :ConcatKDFParams  AlgorithmID = "00"   PartyUInfo = "0
        0"   PartyVInfo = "00" >
15          < ds :DigestMethod  Algorithm = "http://www.w3.org/2001/04/
        xmlenc#sha256"  />
16         </ xenc11 :ConcatKDFParams>
17        </ xenc11 :KeyDerivationMethod>
18        < xenc :OriginatorKeyInfo>
19         < ds :KeyValue>
20          < ds11 :ECKeyValue>
21           < ds11 :NamedCurve  URI = "urn:oid:1.3.36.3.3.2.8.1.1.13"  />
22           < ds11 :PublicKey>BEQ1Qnedbtv...BcGsBDp</ ds11 :PublicKey>
23          </ ds11 :ECKeyValue>
24         </ ds :KeyValue>
25        </ xenc :OriginatorKeyInfo>
26        < xenc :RecipientKeyInfo>
27         < ds :X509Data>
28          < ds :X509Certificate>MIIDJ...X8/jw==</ ds :X509Certificate>
29         </ ds :X509Data>
30        </ xenc :RecipientKeyInfo>
31      </ ds :KeyInfo>
32
33      < xenc :CipherData>
34       < xenc :CipherValue>bSYaL0cXyC...UjDBQ==</ xenc :CipherValue>
35      </ xenc :CipherData>
36     </ xenc :EncryptedKey>
37
38    </ ds :KeyInfo>
39
40
41    < xenc :CipherData>
42     < xenc :CipherValue>pTvKqOqq...kfBb1rw=</ xenc :CipherValue>
43    </ xenc :CipherData>
44   </ xenc :EncryptedData>
45 </ saml2 :EncryptedAssertion>
```

3.4 SAML 2.0 AuthnResponse Assertion Encryption

The standard way of encrypting the SAML Response is to encrypt the Assertions within, as described below.

3.4.1 Assertion encryption support by SAML 2.0

The common type for storing encrypted data is the *EncryptedElementType*. This type is inherited by the *EncryptedAssertion* element of SAML.

3.4.2 Pseudo-code implementation of encryption of SAML Response

1. Check if the encryption functionality is enabled.
2. Set the appropriate data encryption and key encryption algorithms (see section 3.6.4 — *Code snippet – Data & key encryption parameters*.
3. Acquire certificate of the relaying party for symmetric key encryption and use as the key encryption credential (see section 3.6.3 — *Code snippet – Certification credential for encryption*). Note that in eID the entire certificate is conveyed.
4. Clone SAML Response instance to avoid in-place modification of original SAML Response.
5. Generate new symmetric key.
6. Set the *KeyPlacement* INLINE (see section 3.6.6 — *Code snippet – Set up open SAML encrypter*)
7. For each Assertion in the SAML Response,
    a. Encrypt assertion
    b. Add the encrypted assertion to the *EncyptedAssertions* list of the response. (see section *3.6.7 — Code Snippet – Assertion encryption*)
8. Remove all plain Assertion elements from the cloned SAML Response.
9. Return the cloned, encrypted SAML Response (only the SAML Assertion is encrypted).

3.4.3 Pseudo implementation of decryption of SAML Response

1. Check if functionality is enabled.
2. Acquire the single *KeyInfo* of SAML Response if present.(see section 3.6.9 – *Code snippet – Locate & construct the single certificate for decryption in the SAML Response*).
3. Extract X509Certificate from the single *KeyInfo*.
4. Clone SAML Response instance to avoid in-place modification of original SAML Response.
5. Find the appropriate KeyStore entry based on the extracted certificate and retrieve the *PrivateKey* (see section 3.6.10 – *Code snippet – Credential based on the certification for decryption*).
6. For each *EncryptedAssertion* in the SAML Response:
    a. Acquire *EncryptedKey* (symmetric) from the *KeyInfo* of the *EncryptedAssertion*.
    b. Decrypt symmetric key with the *PrivateKey*.
    c. With the decrypted symmetric *SecretKey* decrypt the *EncryptedAssertion* instance. (see section 3.6.11 – *Code snippet – Assertion decryption*).
    d. Add the decrypted Assertion to the Assertions list of the cloned response.
7. Remove all plain *EncryptedAssertion* elements from the cloned SAML Response.
8. Return the cloned, decrypted SAML Response.

3.4.4 Encryption configuration

The encryption can be configured globally for both Proxy Service and Connector instances or individually for each of Proxy Service and Connector instance, therefore, depending on your needs; it may differ from the one shown here.

The first step is to add into the *SAMLEngine.xml* for each module a new configuration for each instance to indicate the configuration files that will be used.

```
<!-- Settings module encryption -->
    < configuration   name = "EncryptionConf" >
        <!-- Specific signature module -->
        < parameter   name = "class"
             value = "eu.eidas.auth.engine.core.impl.EncryptionSW"  />
           <!-- Settings specific module responseTo/FromPointAlias &
requestTo/FromPointAlias parameters will be added -->
        < parameter   name = "fileConfiguration"   value = "EncryptModule_Se
rvice.xml"  />
    </ configuration >
```

The *fileConfiguration* parameter defines the file that contains:

- the keystore;
- the certificates for each country used to encrypt the SAML Responses;
- its own certificate;
- the relative path to the external file that configures the activation of the encryption for each country

This file will be usually allocated inside the module along with the SignModule_<*instance*>.xml files. There will be one per instance.

Inside the file, you need to define a *keyStore* with the trusted certificates for each country.

The property *encryptionActivation* defines the file containing the flags to activate the encryption for the countries. This can be one single file or one file per instance depending on your needs and its complexity. This file is intended to be placed outside the module so configuration changes can be made without rebuilding the whole application.

It contains one property for each country that defines if the Response sent to the country should be encrypted. The decryption process is automatically managed. If the received response is encrypted then it decrypts it, otherwise it does nothing.

EncryptTo.<CountryCode>: Determines if the response sent to the country defined by its country code must be encrypted.

```
<? xml   version = "1.0"   encoding = "UTF-8" ?>
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
 <!-- Activate encryption in the module -->
 < entry   key = "Encryption.enable" >true</ entry >

 < entry   key = "EncryptTo.CA" >false</ entry >
 < entry   key = "EncryptTo.CB" >false</ entry >
 < entry   key = "EncryptTo.CC" >false</ entry >
 < entry   key = "EncryptTo.CD" >false</ entry >
 < entry   key = "EncryptTo.CF" >false</ entry >
</ properties >
```

If some of these parameters are not present the application will work as if the encryption is not activated.

The property *Encryption.enable* is maintained to activate the encryption in the instance itself.

Notes:
1. If encryption and metadata are enabled, the metadata will expose encryption information (see section 4 – *eIDAS Node and SAML Metadata* for more details about metadata). The public key retrieved from metadata will be used for encryption (instead of the one available locally in the keystore).
2. When the configuration parameter **response.encryption.mandatory** is set to true then the settings in encryptionConf.xml are ignored and each response is either encrypted or an error is raised. Error responses are allowed to be unencrypted.

3.4.5 eIDAS SAML 2.0 Encryption example

The following shows an example of encrypting an assertion by replacing plain Assertion (line 5 - 24) with *EncryptedAssertion* (line 25 - 34).

```
1  <? xml  version = "1.0"  encoding = "UTF-8" ?>
2  < saml2p :Response  xmlns:saml2p = "urn:oasis:names:tc:SAML:2.0:protoc
   ol"  xmlns:ds = "http://www.w3.org/2000/09/xmldsig#"  xmlns:saml2 = "
   urn:oasis:names:tc:SAML:2.0:assertion"  xmlns:eidas = "http://
   eidas.europa.eu/saml-extensions"  xmlns:xenc = "http://www.w3.org/
   2001/04/xmlenc#"  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
   Consent = "urn:oasis:names:tc:SAML:2.0:consent:obtained"  Destination
   http://vs-cis-k2:8081/SP/ReturnPage"  ID = "_fb4aed821dbb327caf2aa310
   a6f877bb"  InResponseTo = "_cd5ad7ff3a77529079b39e073597fbc9"
   IssueInstant = "2014-11-20T12:20:00.319Z"  Version = "2.0" >
3   < saml2 :Issuer  Format = "urn:oasis:names:tc:SAML:2.0:nameid-
   format:entity" > http://eidas-connector.gov.xx </ saml2 :Issuer>
4   < saml2p :Status>…</ saml2p :Status>
5   < saml2 :Assertion  ID = "_6a47298ff6092f82d9e540479c46bdfb"
   IssueInstant = "2014-11-20T12:20:00.334Z"  Version = "2.0" >
6    < saml2 :Issuer  Format = "urn:oasis:names:tc:SAML:2.0:nameid-
   format:entity" >  http://eidas-connector.gov.xx  </ saml2 :Issuer>
7    < saml2 :Subject>…</ saml2 :Subject>
8    < saml2 :Conditions  NotBefore = "2014-11-20T12:20:00.334Z"
   NotOnOrAfter = "2014-11-20T12:25:00.319Z" >…</ saml2 :Conditions>
9    < saml2 :AuthnStatement  AuthnInstant = "2014-11-20T12:20:00.334Z" >
10    < saml2 :SubjectLocality  Address = "158.168.60.142"  />
11    < saml2 :AuthnContext>
12     < saml2 :AuthnContextDecl />
13    </ saml2 :AuthnContext>
14   </ saml2 :AuthnStatement>
15   < saml2 :AttributeStatement>
16    < saml2 :Attribute  Name =  http : eidas.europa.eu 1.0 eIdentifier
   NameFormat = "urn:oasis:names:tc:SAML:2.0:attrname-format:uri" >
```

```
17      < saml2 :AttributeValue xmlns:xsi = http : www.w3.org 2001
XMLSchema-instance xsi:type = "xs:anyType" >CA/CA/12345</ saml2 :Attr
ibuteValue>
18    </ saml2 :Attribute>
19    < saml2 :Attribute Name = http : eidas.europa.eu 1.0 givenName
NameFormat = "urn:oasis:names:tc:SAML:2.0:attrname-format:uri" >
20      < saml2 :AttributeValue xmlns:xsi = http : www.w3.org 2001
XMLSchema-instance xsi:type = "xs:anyType" >Javier</ saml2 :Attribute
Value>
21    </ saml2 :Attribute>
22    …
23   </ saml2 :AttributeStatement>
24  </ saml2 :Assertion>
25  < saml2 :EncryptedAssertion>
26   < xenc :EncryptedData Id = "_6a47298ff6092f82d9e540479c46bdfb "
Type = "http://www.w3.org/2001/04/xmlenc#Element" >
27    < xenc :EncryptionMethod Algorithm = "http://www.w3.org/2009/
xmlenc11#aes256-gcm" />
28    < ds :KeyInfo>
29     < xenc :EncryptedKey Id = "_74229d1928a63480c0895c79497d20fe" >
30      < xenc :EncryptionMethod Algorithm = "http://www.w3.org/2001/04/
xmlenc#rsa-oaep-mgf1p" >
31       < ds :DigestMethod Algorithm = "http://www.w3.org/2001/04/
xmlenc#sha256" />
32      </ xenc :EncryptionMethod>
33      < ds :KeyInfo>
34       < ds :X509Data>
35        < ds :X509Certificate>MIIDJzC...8mYfX8/jw==</ ds :X509Certifica
te>
36       </ ds :X509Data>
37      </ ds :KeyInfo>
38      < xenc :CipherData>
39       < xenc :CipherValue>bSYaL0cXyC...UjDBQ==</ xenc :CipherValue>
40      </ xenc :CipherData>
41     </ xenc :EncryptedKey>
42    </ ds :KeyInfo>
43    < xenc :CipherData>
44     < xenc :CipherValue>pTvKqOqq...kfBb1rw=</ xenc :CipherValue>
45    </ xenc :CipherData>
46   </ xenc :EncryptedData>
47  </ saml2 :EncryptedAssertion>
48 </ saml2p :Response>
```

3.4.6 eIDAS SAML 2.0 Encryption and Signature

eIDAS uses XML Signature to verify the authenticity of the SAML messages.
The following rule must apply to the Encryption implementation:

- The signature must be created on the encrypted SAML message (only the SAML Assertion is encrypted).
- The verification of signature must be executed on the encrypted SAML message.

It means that the following pseudo process must be implemented:

1. Construct SAML Response.
2. Encrypt SAML Response.
3. Sign Encrypted SAML Response.
4. Send Encrypted SAML Response to relaying party.
5. Verify the signature of Encrypted SAML Response at the relaying party.
6. If success then decrypt SAML Assertion.
7. Process decrypted SAML Response.

3.4.7 eIDAS SAML 2.0 Encryption with Signature example

```
<? xml  version = "1.0"  encoding = "UTF-8" ?>
< saml2p :Response  xmlns:saml2p = "urn:oasis:names:tc:SAML:2.0:protocol"
xmlns:ds = "http://www.w3.org/2000/09/xmldsig#"  xmlns:saml2 = "urn:oasis:n
ames:tc:SAML:2.0:assertion"  xmlns:eidas = "http://eidas.europa.eu/saml-
extensions"  Consent = "urn:oasis:names:tc:SAML:2.0:consent:obtained"
Destination = "http://localhost:8080/eidasNode/SpecificIdPResponse"  ID = "
_f38631b536398fb5e9432e91bb7f764d"  InResponseTo = "_9884b5a27102a5870455be9
19d126faa"  IssueInstant = "2014-12-23T10:32:12.751Z"  Version = "2.0" >
 < saml2 :Issuer  Format = "urn:oasis:names:tc:SAML:2.0:nameid-
format:entity" >  http://eidas-connector.gov.xx  </ saml2 :Issuer>
 < ds :Signature>
  < ds :SignedInfo>
   < ds :CanonicalizationMethod  Algorithm = "http://www.w3.org/2001/10/xml-
exc-c14n#"  />
   < ds :SignatureMethod  Algorithm = "http://www.w3.org/2000/09/
xmldsig#rsa-sha1"  />
   < ds :Reference  URI = "#_f38631b536398fb5e9432e91bb7f764d" >
    < ds :Transforms>
     < ds :Transform  Algorithm = "http://www.w3.org/2000/09/
xmldsig#enveloped-signature"  />
     < ds :Transform  Algorithm = "http://www.w3.org/2001/10/xml-exc-c14n#"
/>
    </ ds :Transforms>
    < ds :DigestMethod  Algorithm = "http://www.w3.org/2000/09/xmldsig#sha1"
/>
    < ds :DigestValue>bcWvl4NOSgKmVZglli9SKRKJipE=</ ds :DigestValue>
```

```
    </ ds :Reference>
   </ ds :SignedInfo>
   < ds :SignatureValue>PdwB/…IW+yg==</ ds :SignatureValue>
   < ds :KeyInfo>
    < ds :X509Data>
     < ds :X509Certificate>MIIDJzC…YfX8/jw==</ ds :X509Certificate>
    </ ds :X509Data>
   </ ds :KeyInfo>
  </ ds :Signature>
  < saml2p :Status>
   < saml2p :StatusCode  Value = "urn:oasis:names:tc:SAML:2.0:status:Success"
/>
   < saml2p :StatusMessage> urn:oasis:names:tc:SAML:2.0:status:Success </
saml2p :StatusMessage>
  </ saml2p :Status>
  < saml2 :EncryptedAssertion>
   < xenc :EncryptedData  xmlns:xenc =  http : www.w3.org 2001 04 xmlenc#  Id =
_485c8629476d743eb85c244ac3f113f0"   Type = "http://www.w3.org/2001/04/
xmlenc#Element" >
    < xenc :EncryptionMethod  Algorithm = "http://www.w3.org/2009/
xmlenc11#aes256-gcm"  />
    < ds :KeyInfo>
     < xenc :EncryptedKey  Id = "_74229d1928a63480c0895c79497d20fe" >
      < xenc :EncryptionMethod  Algorithm = "http://www.w3.org/2001/04/
xmlenc#rsa-oaep-mgf1p" >
       < ds :DigestMethod  Algorithm = "http://www.w3.org/2001/04/
xmlenc#sha256"  />
      </ xenc :EncryptionMethod>
      < ds :KeyInfo>
       < ds :X509Data>
        < ds :X509Certificate>MII…8mYfX8/jw==</ ds :X509Certificate>
       </ ds :X509Data>
      </ ds :KeyInfo>
      < xenc :CipherData>
       < xenc :CipherValue>bS…jDBQ==</ xenc :CipherValue>
      </ xenc :CipherData>
     </ xenc :EncryptedKey>
    </ ds :KeyInfo>
    < xenc :CipherData>
     < xenc :CipherValue>pTvKq…Bb1rw=</ xenc :CipherValue>
    </ xenc :CipherData>
   </ xenc :EncryptedData>
  </ saml2 :EncryptedAssertion>
</ saml2p :Response>
```

3.5 XML Encryption/Decryption implementation

3.5.1 OpenSAML - XML Tooling

Currently eIDAS uses OpenSAML signature utilities. The XML Encryption/Decryption engine of Open SAML named xmltooling implements the necessary AES-GCM algorithms: http://www.w3.org/2009/xmlenc11#aes256-gcm is strongly recommended to use.

In order to be able to implement large key size encryption refer to grepcode.com for the details of implemented algorithms: http://grepcode.com/file/repo1.maven.org/maven2/org.opensaml/xmltooling/1.4.1/org/opensaml/xml/encryption/EncryptionConstants.java?av=f

For implementation details refer to: https://wiki.shibboleth.net/confluence/display/OpenSAML/OSTwoUserManJavaXMLEncryption

An advantage of xmltooling is that it provides SAML Specific Encrypter and Decrypter for Assertions besides the generic XML Encryption. **The generic XML Encryption of xmltooling can be used.**

3.5.2 Component dependencies

The following diagram illustrates the dependencies of the various components in an eIDAS implementation.

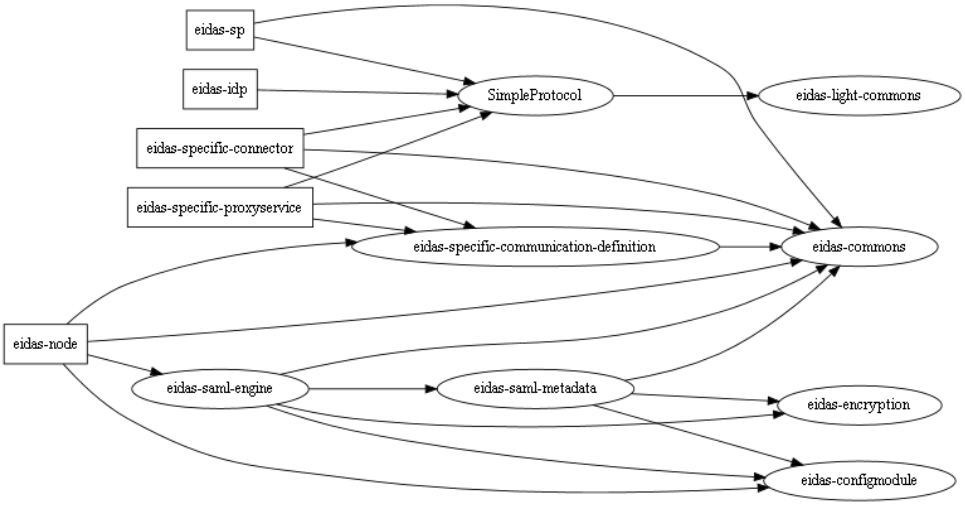

Figure 2: Component dependencies

3.5.3 Code snippet – Certification credential for encryption

```
samlAuthnResponseEncrypter = new SAMLAuthnResponseEncrypter();
…
// Load certificate matching the configured alias
X509Certificate responsePointAliasCert = (X509Certificate)
encryptionKeyStore.getCertificate(alias);
```

```
// Create basic credential and set the EntityCertificate
BasicX509Credential credential = new BasicX509Credential();
credential.setEntityCertificate(responsePointAliasCert);
// Execute encryption
samlAuthnResponseEncrypter.encryptSAMLResponse(authResponse, credential);
```

3.5.4 Code snippet – Data & key encryption parameters

```
// Set Data Encryption parameters
EncryptionParameters encParams = new EncryptionParameters();
encParams.setAlgorithm(…);
// Set Key Encryption parameters
KeyEncryptionParameters kekParams = new KeyEncryptionParameters();
kekParams.setEncryptionCredential(credential);
kekParams.setAlgorithm(…);
KeyInfoGeneratorFactory kigf =
Configuration.getGlobalSecurityConfiguration().getKeyInfoGeneratorManager().g
etDefaultManager().getFactory(credential);
kekParams.setKeyInfoGenerator(kigf.newInstance());
```

3.5.5 Code snipped – ECDH Key encryption parameters

```
// Set Key Encryption parameters
ECDHKeyAgreementParameters kekParams = new ECDHKeyAgreementParameters();
kekParams.setPeerCredential(credential);
kekParams.setKeyInfoGenerator(ExtendedDefaultSecurityConfigurationBootstrap
    .buildDefaultKeyAgreementKeyInfoGeneratorFactory().newInstance());
```

3.5.6 Code snippet – Set up open SAML encrypter

```
// Setup Open SAML Encrypter
Encrypter samlEncrypter = new Encrypter(encParams, kekParams);
samlEncrypter.setKeyPlacement(Encrypter.KeyPlacement.INLINE);
```

3.5.7 Code Snippet – Assertion encryption

```
samlResponseEncryptee = XMLObjectHelper.cloneXMLObject(samlResponse);
…
for  (Assertion : samlResponseEncryptee.getAssertions()) {
    manageNamespaces(assertion);
    EncryptedAssertion = samlEncrypter.encrypt(assertion);
    samlResponseEncryptee.getEncryptedAssertions().add(encryptedAssertion);
}
samlResponseEncryptee.getAssertions().clear();
```

3.5.8 Code snippet – Manage specific namespace prefix

Necessary when the SAML Response XML does not use the "saml:" prefix. In this case the decryption will not be able to identify the unknown prefix of the detached Assertion element unless the following is applied!

```
Set<Namespace> namespaces = assertion.getNamespaceManager().getNamespaces();
for  (Namespace : namespaces) {
     if  ( "urn:oasis:names:tc:SAML:2.0:assertion" .equals(namespace.getName
spaceURI()) && assertion.getDOM().getAttributeNode( "xmlns:"  +
namespace.getNamespacePrefix()) ==  null ){

assertion.getNamespaceManager().registerNamespaceDeclaration(namespace);
        assertion.getDOM().setAttribute( "xmlns:"  +
namespace.getNamespacePrefix(), namespace.getNamespaceURI());
    }
}
```

3.5.9 Code snippet – Locate & construct the single certificate for decryption in the SAML Response

The following snippet assumes that the encryption was applied as shown in the above configuration.

```
EncryptedAssertion encAssertion =
authResponse.getEncryptedAssertions().get( 0 );
EncryptedKey encryptedSymmetricKey =
encAssertion.getEncryptedData().getKeyInfo().getEncryptedKeys().get( 0 );
org.opensaml.xml.signature.X509Certificate keyInfoX509Cert =
encryptedSymmetricKey.getKeyInfo().getX509Datas().get( 0
).getX509Certificates().get( 0 );
```

```
final ByteArrayInputStream bis = new
 ByteArrayInputStream(Base64.decode(keyInfoX509Cert.getValue()));
final CertificateFactory certFact = CertificateFactory.getInstance( "X.509" )
;
final X509Certificate keyInfoCert = (X509Certificate)
certFact.generateCertificate(bis);
```

3.5.10 Code snippet – Credential based on the certification for decryption

```
samlAuthnResponseDecrypter = new SAMLAuthnResponseDecrypter();
for ( final Enumeration<String> e = encryptionKeyStore.aliases();
e.hasMoreElements();) {
    aliasCert = e.nextElement();
    responsePointAliasCert = (X509Certificate)
encryptionKeyStore.getCertificate(aliasCert);
     // Check if certificates equal
     if (Arrays.equals(keyInfoCert.getTBSCertificate(),responsePointAliasCer
t.getTBSCertificate())) {
        alias = aliasCert;
         break ;
    }
} // Handle if certificate not found… etc.
…
// Get PrivateKey by found alias final
PrivateKey responsePointAliasPrivateKey = (PrivateKey)
encryptionKeyStore.getKey(alias, properties.getProperty( "keyPassword" ).toCh
arArray());
// Create basic credential and set the PrivateKey
BasicX509Credential credential = new BasicX509Credential();
credential.setPrivateKey(responsePointAliasPrivateKey);
// Execute decryption
samlAuthnResponseDecrypter.decryptSAMLResponse(authResponse, credential);
```

3.5.11 Code snippet – Assertion decryption

```
samlResponseDecryptee =
XMLObjectHelper.cloneXMLObject(samlResponseEncrypted);
…
for (EncryptedAssertion
encAssertion :samlResponseDecryptee.getEncryptedAssertions()) {
```

```
    DecryptionParameters decryptionParameters =  new
 DecryptionParameters();
    DecryptionConfiguration config =
DefaultSecurityConfigurationBootstrap.buildDefaultDecryptionConfiguration();

decryptionParameters.setDataKeyInfoCredentialResolver( config.getDataKeyInfoC
redentialResolver());

    ChainingEncryptedKeyResolver encryptedKeyResolver =  new
 ChainingEncryptedKeyResolver(
        Arrays.asList(
             new  InlineEncryptedKeyResolver(),
             new  EncryptedElementTypeEncryptedKeyResolver(),
             new  SimpleRetrievalMethodEncryptedKeyResolver()
        )
    );
    decryptionParameters.setEncryptedKeyResolver(encryptedKeyResolver);

    List<KeyInfoProvider> providers = Arrays.asList(
         new  KeyAgreementMethodKeyInfoProvider(),
         new  RSAKeyValueProvider(),
         new  InlineX509DataProvider()
    );
    List<KeyInfoCredentialResolver> KeyInfoResolverList = Arrays.asList(
         new  LocalKeyInfoCredentialResolver(
            providers,
             new  CollectionKeyInfoCredentialResolver(
                Arrays.asList(credentials)
             )
        )
    );
    decryptionParameters.setKEKKeyInfoCredentialResolver( new
 ChainingKeyInfoCredentialResolver(keyInfoResolverList));
    Decrypter dataDecrypter =  new  Decrypter(decryptionParameters);
    dataDecrypter.setRootInNewDocument( true );
    Assertion = dataDecrypter.decrypt(encAssertion);
    samlResponseDecryptee.getAssertions().add(assertion);
}
samlResponseDecryptee.getEncryptedAssertions().clear();
```

3.6 Sources of further information

The following sources provide further information on the SAML XML encryption standard and its implementation.

Symmetric key encryption

http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p with 2048 key size.

XMLEnc XSD

http://www.w3.org/TR/2013/PR-xmlenc-core1-20130124/xenc-schema.xsd

4 eIDAS Node and SAML Metadata

4.1 Presentation

SAML metadata are configuration data required to automatically negotiate agreements between system entities, comprising:

- Identifiers;
- binding support and endpoints;
- certificates;
- keys;
- cryptographic capabilities; and
- security and privacy policies.

SAML metadata is standardised by OASIS and signed XML data (signing SAML metadata is not mandatory but is assumed in this document). SAML metadata can be provided as a metadata file at a URL, as indirect resolution through DNS, or by other means.

The advantage of SAML metadata is that it is part of the SAML specifications, thus already closely related to the planned eIDAS interoperability standard using SAML as message format and exchange protocol. It is supported by SAML implementations and libraries.

The infrastructure data needed is twofold:

- notification-related and security data e.g. authentication online; and
- sole technical data e.g. supported protocol versions.

Both are needed for a functioning infrastructure. The distinction above is made, as the issuance and security relevance may be different. The former (notification-related data) is information provided by the MS that also relates to parties being liable under eIDAS art. 11. The latter (sole technical data) relates to the functioning of components.

See Table 1 for an indicative list of metadata.

4.2 Use cases

4.2.1 Identification of eIDAS-Nodes

To provide an uninterrupted chain of trust for authentications, as well as an uninterrupted chain of responsibility for integrity/authenticity and confidentiality for personal identification data, eIDAS-Nodes should be securely identified before transmitting data to them or accepting data from them.

4.2.1.1 Proxy-based schemes

Certificates for SAML signing and encryption of messages between Connector and Proxy Service are exchanged via SAML Metadata.

4.2.1.2 Middleware-based schemes

Certificates for SAML signing and encryption of messages between an eIDAS-Node Connector and eIDAS-Middleware Services are unsigned, therefore they do not need to be verified. They are exchanged directly between

the entities, since there is a one-to-one correspondence between an eIDAS-Node Connector and an eIDAS Middleware-Service.

4.2.2 Request messages verification

Each eIDAS-Node Proxy Service should verify the integrity/authenticity of a SAML Request message before processing the request.

For eIDAS-Proxy-Services, this comprises the following steps:

1. Retrieve the SAML Metadata object of the sender from the metadata URL contained in the request or from a local cache. A local cache is used to reduce latency (redirect processing is usually faster than POST-processing).
2. Verify the signature of the SAML Metadata object including Certification Path Validation according to \[RFC5280\]. The Path should start at a valid trust anchor.
3. Extract the signature certificate of the sender from the SAML Metadata and use them to verify the signature of the SAML Request message.

Request messages which cannot be verified via this procedure should be rejected.

4.2.3 Response messages verification

Each eIDAS-Connector should verify the authenticity of a SAML Response message before processing the included assertion.

For messages originating at an eIDAS-Proxy Service, this comprises the following steps:

1. Retrieve the SAML Metadata object of the sender, either from the metadata URL contained in the Assertion or from a local cache. A local cache is used to reduce latency (redirect processing is usually faster than POST-processing).
2. Verify the signature of the SAML Metadata object including Certification Path Validation according to \[RFC5280\]. The Path should start at a valid trust anchor.
3. Extract the signature certificate of the sender from the SAML Metadata and using it to verify the signature of the SAML Response message.
4. If the SAML Assertion is signed, its signature MAY be verified.

Assertion messages which cannot be verified via this procedure should be rejected. Unsolicited Response Messages should not be accepted.

See Table 2 for a list of metadata related parameters.

4.2.4 Metadata exchange

To provide an uninterrupted chain of trust for authentications, as well as an uninterrupted chain of responsibility for integrity/authenticity and confidentiality for personal identification data, eIDAS-Nodes must be securely identified. The Architecture defines two communication relationships:

- Communication between eIDAS Connectors and Proxy-Services; and
- Communication between eIDAS Connectors and Middleware-Services.

For Middleware-Services, there is a one-to-one relationship between the Connector and the Middleware-Service; identification is done directly, e.g. via self-signed certificates. Therefore, this section is only concerned with exchanging metadata for Connectors and Proxy-Services, although direct exchange can also be done using the same mechanism.

Metadata exchange is based on the following principles:

- The trust anchors for all eIDAS-Nodes are the MS's. No central trust anchor is provided. Trust Anchors are exchanged bilaterally between MS's.
- Metadata are distributed in the form of SAML Metadata [SAML-Meta]. Metadata objects are signed by the Trust Anchor or by another entity (e.g. the operator of the eIDAS-Node) authorised via a certificate chain starting from the trust anchor.

4.2.4.1 Trust anchor

All MSs should bilaterally exchange trust anchors in the form of certificates, certifying a signing key held by the MS (the 'Root'). This signing key can either be used:

- to directly sign SAML metadata objects; or
- as root certificate of a PKI used to sign SAML metadata objects.

"Certificates of the root and subordinate certificates SHALL follow [RFC5280]" Extract from section 6.2 in *eIDAS – Interoperability Architecture* https://ec.europa.eu/digital-building-blocks/wikis/download/attachments/82773108/eIDAS%20Interoperability%20Architecture%20v.1.2%20Final.pdf.

MS should ensure that all SAML metadata objects signed directly or indirectly under this Root describe valid eIDAS-Nodes established in that MS.

4.2.4.2 SAML metadata

The eIDAS-Connector provides metadata about the Connector/Proxy Service in the form of SAML Metadata (see *SAML-Meta* in section *7 — References*).

SAML Metadata objects are signed and include a certificate chain starting at a trust anchor and terminating with a certificate certifying the key used to sign the Metadata object.

The SAML Metadata Interoperability Profile Version should be followed (see *SAML-MetaIOP* in section *7 — References*). Certificates should be encapsulated in X509Certificate-elements.

The current version of the eIDAS-Node publishes metadata information under the *EntityDescriptor* element. It may read metadata information available under either:

- *EntityDescriptor* element; or
- *EntitiesDescriptor* element data from a static file (aggregate metadata).

4.2.4.3 Metadata location

The SAML Metadata are only available under an HTTPS URL.

SAML Requests and Responses contain an HTTPS URL in the <Issuer> element pointing to the SAML metadata object of the issuer of the request/assertion (see section *4.1.1 of SAML-Meta* 'Well-Known Location' method in section *7 — References*).

4.2.4.4 Metadata verification

For verification of SAML metadata, verifiers build and verify a certificate path according to RFC5280 (see section *7 — References*) starting from a trusted Trust Anchor (see section *4.2.4.1 — Trust anchor*) and ending at the signer of the metadata object. Revocation checks are performed for all certificates containing revocation information.

All restrictions contained in the metadata object (e.g. validity period) are honoured by the verifier.

4.3 Message format

4.3.1 Metadata in SAML Requests & SAML Responses

SAML Requests and Assertions (SAML Responses) contain an HTTPS URL in the <Issuer> element pointing to the SAML Metadata object of the issuer of the request/assertion (see section *4.1.1 of SAML-Meta*: Metadata for the OASIS Security Assertion Markup Language at https://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf).

4.3.2 Metadata profile for eIDAS-Nodes

The profile contains the following information:

- the MS operating the eIDAS-Node;
- the URL under which the eIDAS-Node is operated;
- a communication address (preferably email);
- the certificates corresponding to the SAML signature and decryption keys; and
- an indication if the eIDAS-Node serves public and/or private parties.

4.3.3 List of eIDAS metadata

Table 1: List of eIDAS metadata

Metadata	SAML MD	Comment
Validity period	validUntil	Indicates expiration date of metadata elements
Protocol version(s)	protocolSupport-Enumeration	For multi-protocol support, this is likely to change and be amended over time
Supported NameID	SSODescriptor NameIDFormat	In case several NameID formats can get requested
Supported attributes	SSODescriptor Attribute; EntityAttribute extensions	At least the minimum data set and available matching data. Should also allow for indicating additional sector-specific attributes.
SAML signer	KeyDescriptor	Certificate to sign SAML requests or responses
Encryption certificate	KeyDescriptor	Certificate to encrypt a SAML Response

Metadata	SAML MD	Comment
Supported LoA	SAML Extensions EntityAttributes AttributeName http://eidas.europa.eu/LoA	Automatically retrieving which LoAs are supported allows a component (relying party, eIDAS-Node Connector, V-IDP) to only redirect to MS that also offer the LoA needed by a relying party
Organization info	OrganizationName, organizationDisplayName, OrganizationURL	Organization responsible for an entity
Contact info	ContactPersonType, ContactPersonGivenName, ContactPersonSurname, ContactPersonEmail, ContactPersonPhone	Support contact for an entity (e.g. eIDAS-Node, V-IDP)
Requester Id Flag	Attribute Name="http://macedir.org/entity-category" AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"	Flag that when present indicates to the Relying Party (I.e. Connector) that the Requester Id is to be sent in the eIDAS Request for the case of private SP.

4.4 Details of the metadata used in the eIDAS-Node

The following validations are made on the metadata (when receiving a message):

- verify validity duration (see *metadata.validity.duration* parameter);
- check the metadata signature (trusted certificate loaded in the keystore – see *metadata.check.signature* parameter, not applicable to Middleware Services);
- check the *assertionConsumerServiceURL* is the same between the metadata and the message;

4.4.1 Validation of supported Levels of Assurance

The supported levels of assurance are validated at the connector side when building the eIDAS SAML Request and when receiving the eIDAS SAML Response.

When building the eIDAS SAML Request, it is verified that:

If a notified level of assurance is requested, the requested notified level requested is compared to the notified levels of assurance published in the metadata of the target ProxyService.

If the proxy service is not publishing in its metadata any equal of higher notified level of assurance compare to the requested notified levels of assurance or if no notified level of assurance is requested, then it is checked if there is at least one not-notified level of assurance published in the metadata of the target ProxyService that is equal to a requested not-notified levels of Assurance.

If no match was found for the two previous check then an error is returned.

When receiving the eIDAS SAML Response, it is verified that:

The response level of assurance is either a notified level of assurance which is of an equal or higher value of the one requested or a not-notified level of assurance which is equal to a requested level of assurance.

4.4.2 Support of dynamic and cached use of metadata

The eidas.xml entry *metadata.http.retrieval* activates the retrieval of metadata information using http/https (by default or value "true"). The *metadata.file.repository* entry defines a folder that may contain the static SAML metadata to be used. The application will process files contained in the folder that are well-formed XML and have .xml extension. Other files will be ignored but logging will be performed when this occurs. If the folder does not exist or the parameter refers to an unavailable location, only dynamically retrieved metadata may be used. If the static metadata is expired, the application will try to reach the remote location. Please check **Table 2** for more details on these two entries.

Table 2: Metadata related parameters

Key	Description
metadata.activate	Activate or not the publishing of the SAML metadata from the Connector and ProxyService. Publishing of the metadata is activate by default.
metadata.file.repository	Path to folder where static SAML metadata configuration is stored. If static SAML metadata for a given url exists in this folder, it will be used and it will override the retrieval of new remote metadata using HTTP otherwise if it does not exist or refers an unavailable location, only dynamically retrieved metadata may be used.
metadata.http.retrieval	Activate the retrieval of metadata information using http/https. Default is 'true'. When set on 'false', only static metadata may be used.
metadata.restrict.http	Disable http for metadata retrieval (such that only https will be allowed)
tls.enabled.protocols	The SSL/TLS protocols to be used when retrieving metadata via https. The default values, as by eIDAS specification, are: TLSv1.2.
tls.enabled.ciphers	The SSL/TLS cipher suites to be used when retrieving metadata via https. Empty value means 'use provided JDK cipher suites'.
metadata.sector	Value of SPType (public or private) to be published in the metadata (see *eIDAS Technical Specifications)*
metadata.node.country	Value of the Node Country to be published in the metadata Value must be compliant with the ISO 3166-1 alpha-2 format. (see eIDAS Technical Specifications for more information)

Key	Description
metadata.check.signature	When set to 'false', the check of metadata signature is disabled. (Note: by default metadata signature validation is active)
metadata.validity.duration	Duration of validity for dynamic metadata (in seconds). Default is 86400 (one day)

requester.id.flag	Requester Id flag, when set to true enables publishing of requester Id entity category attribute in Proxy-service's metadata. This attribute will not be published when the flag is not set, or if the flag is set to false.
	
	When eiDAS Request contains the RequesterID element it will be e.g. as follows:
	
	`</ saml2p :AuthnRequest>`
	`.....`
	`< saml2p :Scoping>`
	`< saml2p :RequesterID> http://localhost:8080/SP </ saml2p :RequesterID>`
	`</ saml2p :Scoping>`
	`</ saml2p :AuthnRequest>`

The above table of parameters concerns only Node parameters, and default values, if defined, are defined in the the default eidas.xml. See section 4.3 of the *eIDAS-Node Installation and configuration guide* for more information.

4.4.3 Internal cache behaviour

When a piece of metadata is requested for processing (e.g. when an incoming request is processed by eIDAS), the validity is checked. An error will be printed in the logs if the metadata has expired.

A statically loaded metadata (if *metadata.http.retrieval* set to false) will not be replaced by a piece of metadata retrieved through http/https.

Metadata is loaded only when the application starts, i.e. eIDAS-Node does not support key rollover.

4.4.4 Parametrization of the metadata signing certificate and trust chain

The certificate used to sign SAML messages is not the same as that used to sign the metadata. All MS should bilaterally exchange trust anchors in the form of certificates, certifying a signing key held by the MS (the "Root"). This signing key can either be used to directly sign SAML metadata objects, or as root certificate of a PKI used to sign SAML metadata objects.

Each communication point in the eIDAS Node needs to be configured to use a specific certificate to sign its metadata. It is configured into the same file used to configure the encryption (see section 5.3 — *Configuration* for

signature configuration information).This file will be usually allocated inside the module along with the SignModule_<*instance*>.xml files. This file should contain the following lines:

```
 <? xml   version = "1.0"   encoding = "UTF-8" ?>
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < comment >SWModule sign with PKCS12.</ comment >
    < entry   key = "keyStorePath" >keystores/keycountry1.p12</ entry >
    < entry   key = "keyStorePassword" >passkey1</ entry >
    < entry   key = "keyPassword" >pass1</ entry >
    < entry   key = "issuer" >CN=eidas, C=ES</ entry >
    < entry   key = "serialNumber" >4BA0BD66</ entry >
    < entry   key = "keyStoreType" >PKCS12</ entry >
    <!—Metadata signature configuration -->
    < entry   key = "metadata.keyStorePath" >eidasKeyStore_METADATA.p12</
entry >
    < entry   key = "metadata.keyStorePassword" >keystorepassword</ entry >
    < entry   key = "metadata.keyPassword" >keypassword</ entry >
    < entry   key = "metadata.issuer" >issuerDN</ entry >
    < entry   key = "metadata.serialNumber" >serialNumber</ entry >
    < entry   key = "metadata.keyStoreType" >PKCS12</ entry >
</ properties >
```

If a parameter is missing, the corresponding default parameter's value will be used. E.g. if *metadata.keyStorePath* is not set, then the value from *keyStorePath* parameter is used. However, in this case, the keystore keycountry1.p12 will be used as a source of the certificates to be published in the metadata of the eIDAS Node, which might originate publishing of non-intended certificates in metadata URLs.

Therefore, the correct configuration must contain *metadata.keyStorePath* referencing a different keystore and related entries as presented above. This keystore must contain, at least, the key that signs the metadata. The other party that consumes the metadata, will need, in this case, to trust the corresponding certificate to that key.

In the case of using the trust chain functionalities, the *metadata.keyStorePath* will also include the certificates composing of the trust chain as trusted entries. It must not contain any other certificates or keys; otherwise it will be published in the metadata of the eIDAS Node.

If, by mistake, a certificate not belonging to the trust chain is added as a trusted entry to the metadata keystore, the metadata produced will show it. Even if the consumer of this metadata trusts that certificate, the trust chain will not be validated. However, this is to be avoided since it will give a wrong impression to the party that consumes the metadata that it can trust that certificate. So ideally, the keystore should only contain the certificates that belong to the trust chain of the key signing related certificate.

Note that if the root certificate is not published, but the consumer party trusts it, the certificate path will be successfully validated. Of course, all the certificates up to the root must be published otherwise the trust chain will not be validated.

Due to the temporary interoperability with eIDAS-Node 1.4.x requirement, the order of published certificates in the trust chain is set firstly by the signing certificate on top and subsequently by the issuer certificates. Being last means being the highest CA certificate in the trust chain contained in the keystore.

5 eIDAS-Node Protocol Engine

5.1 Introduction

The eidas-saml-engine-X.X.jar library is an eIDAS module that is used by any module that needs to generate or validate SAML messages with eIDAS specification. It is also possible to customise the behaviour with custom implementation of *ProtocolProcessor* to create generic SAML messages without eIDAS features.

5.2 Dependencies

This library is based on OpenSAML (version 4.1.1).

Additionally, it is dependent on *eidas-saml-metadata-xx.jar*, *eidas-commons-XX.jar and eidas-encryption*.

5.3 Configuration

It is possible to create many instances of *ProtocolEngine* within one application. This makes possible the basic use of two different *ProtocolProcessor* instances in the eIDAS-Node, one eIDAS for the eIDAS Network and one custom for the MS-Specific part. In case the eIDAS-Node is operated as both eIDAS-Node Proxy Service and eIDAS-Node Connector, it means four different *ProtocolEngine* instances.

SAMLEngine.xml**:** configuration file, where the instances are defined. The actual filename can be different, passed to a *ProtocolEngineFactory*, along with a *defaultPath* element. The default location in the eIDAS-Node implementation is set by *EIDAS_CONFIG_REPOSITORY* or by *SPECIFIC_CONFIG_REPOSITORY* environment values.

Please note that there can be more than one configuration file with more than one engine configuration. In addition there can be multiple Factories.

```
<!-- Configuration name-->
< instance   name = "CONF1" >
    <!-- Configurations parameters SamlEngine  -->
    < configuration   name = "SamlEngineConf" >
        < parameter   name = "fileConfiguration"   value = "SamlEngine_Conf1
.xml"  />
    </ configuration >

    <!-- Settings module signature-->
    < configuration   name = "SignatureConf" >
        <!-- Specific signature module -->
        < parameter   name = "class"   value = "eu.eidas.auth.engine.core.im
pl.SignSW"  />
        <!-- Settings specific module -->
        < parameter   name = "fileConfiguration"   value = "SignModule_Conf1
.xml"  />
    </ configuration >
</ instance >
<!-- Settings module encryption -->
```

```
< configuration   name = "EncryptionConf" >
    <!-- Specific encryption module -->
    < parameter   name = "class"   value = "eu.eidas.auth.engine.core.impl.E
ncryptionSW"  />
    <!-- Settings specific module -->
    < parameter   name = "fileConfiguration"   value = "EncryptModule_Conf1.
xml"  />
</ configuration >
<!-- Settings for the ExtensionProcessor module -->
< configuration   name = "ProtocolProcessorConf" >
    <!-- Specific ExtensionProcessor module -->
    < parameter   name = "class"   value = "eu.eidas.sp.SpEidasProtocolProce
ssor"  />
    < parameter   name = "coreAttributeRegistryFile"   value = "saml-engine-
eidas-attributes.xml"  />
    < parameter   name = "additionalAttributeRegistryFile"   value = "saml-
engine-additional-attributes.xml"  />
    < parameter   name = "metadataFetcherClass"   value = "eu.eidas.sp.metad
ata.SPCachingMetadataFetcher"  />
</ configuration >
```

All internal path elements are relative.
Each engine-instance needs four configuration steps:

1. SAML message configuration for eIDAS (*SamlEngineConf*).
2. Configuration of the Sign and Validation Module (*SignatureConf*).
    a. Specific Sign Module (class) and its configuration file (*fileConfiguration*).
    b. To create a new Sign and Validation module, it must implement the *SAMLEngineSignI* interface (*eu.eidas.auth.engine.core.SAMLEngineSignI*).
    c. The Sign module configuration file must be an xml file.
3. Configuration of the Encryption Module (*EncryptionConf*).
    a. Specific Encryption Module (class) and its configuration file (*fileConfiguration*).
    b. To create a new Encryption module, it must implement the SAMLEngineEncryptionI interface (*eu.eidas.auth.engine.core.SAMLEngineEncryptionI*).
    c. The Encryption module configuration file must be an xml file.
4. Configuration of the ProtocolProcessor Module:
    a. Attribute Registry configuration files. These are xml files or hardcoded definitions.
    b. The files contain the eIDAS compliance attributes.
    c. If needed, an additional attribute file is provided to allow declaration of sector specific attributes.

The following are the possible options to configure at the saml-engine behaviour when generating and validating attributes (SamlEngine_Conf1.xml):

```
<!-Types of consent obtained from the user for this authentication and data
transfer.Allow values: 'unspecified'.->
< entry   key = "consentAuthnRequest" >unspecified</ entry >
```

```
<!—Allow values: 'obtained', 'prior', 'current-implicit', 'current-
explicit', 'unspecified'.->
< entry   key = "consentAuthnResponse" >obtained</ entry >

<!—URI representing the classification of the identifier. Allow values:
'entity'.-->
< entry   key = "formatEntity" >entity</ entry >

<!—The SOAP binding is only supported for direct communication between SP-MW
and VIdP-->
< entry   key = "protocolBinding" >HTTP-POST</ entry >

<!—-A friendly name for the attribute that can be displayed to a user -->
< entry   key = "friendlyName" >false</ entry >

<!-Optional attributes->
< entry   key = "eIDSectorShare" >false</ entry >
< entry   key = "eIDCrossSectorShare" >false</ entry >
< entry   key = "eIDCrossBorderShare" >false</ entry >

<!—Attributes with require option. Set this to true if you want to support
optional values-->
< entry   key = "isRequired" >true</ entry >

<!-Subject cannot be confirmed on or after this number of seconds(positive
number)->
< entry   key = "timeNotOnOrAfter" >300</ entry >

<!-Validation IP of the response->
< entry   key = "ipAddrValidation" >false</ entry >

<!-One time use->
< entry   key = "oneTimeUse" >true</ entry >
```

Note that for the saml engine used by the EIDAS-Connector, the type of consent is the following:

```
<!-Types of consent obtained from the user for this authentication and data
transfer.Allow values: 'unspecified'.->
< entry   key = "consentAuthnRequest" >unspecified</ entry >
```

Note that for the saml engine used by the EIDAS-ProxyService, the type of consent is the following:

```
<!--Types of consent obtained from the user for this authentication and data
```

```
<!—Allow values: 'obtained', 'prior', 'current-implicit', 'current-
explicit', 'unspecified'.->
< entry   key = "consentAuthnResponse" >obtained</ entry >
```

If the sign module is configured with *eu.eidas.auth.engine.core.impl.SignSW* it needs to configure the file (SignModule_Conf1.xml):

```
<? xml   version = "1.0"   encoding = "UTF-8" ?>
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < comment >SWModule sign with PKCS12.</ comment >
    < entry   key = "keyStorePath" >keystores/keycountry1.p12</ entry >
    < entry   key = "keyStorePassword" >passkey1</ entry >
    < entry   key = "keyPassword" >pass1</ entry >
    < entry   key = "issuer" >CN=eidas, C=ES</ entry >
    < entry   key = "serialNumber" >4BA0BD66</ entry >
    < entry   key = "keyStoreType" >PKCS12</ entry >
    <!—Metadata signature configuration -->
    < entry   key = "metadata.keyStorePath" >eidasKeyStore__METADATA.p12</
entry >
    < entry   key = "metadata.keyStorePassword" >keystorepassword</ entry >
    < entry   key = "metadata.keyPassword" >keypassword</ entry >
    < entry   key = "metadata.issuer" >issuerDN</ entry >
    < entry   key = "metadata.serialNumber" >serialNumber</ entry >
    < entry   key = "metadata.keyStoreType" >PKCS12</ entry >
</ properties >
```

If the encryption module is configured with *eu.eidas.auth.engine.core.impl.EncryptionSW* it needs to configure the file (EncryptModule_Conf1.xml):

```
<? xml   version = "1.0"   encoding = "UTF-8" ?>
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < entry   key = "keyStorePath" >keystores/keycountry1.p12</ entry >
    < entry   key = "keyStorePassword" >passkey1</ entry >
    < entry   key = "keyPassword" >pass1</ entry >
    < entry   key = "issuer" >CN=eidas, C=ES</ entry >
    < entry   key = " serialNumber" >4BA0BD66</ entry >
    < entry   key = "keyStoreType" >PKCS12</ entry >
    <!-- Management of the encryption activation -->
    < entry   key = "encryptionActivation" >encryptionConf.xml</ entry >
```

```
    < entry   key = "responseToPointIssuer.BE" >CN=local-demo-cert,
OU=DIGIT, O=European Commission, L=Brussels, ST=Belgium,C=BE</ entry > < entr
y   key = "responseToPointSerialNumber.BE" >54C8F779</ entry >
    <!-- … other requesters to which the responses will be encrypted …- - >
    <!-- Key transport config -->
    <!—private key to be used for decryption-->
    < entry   key = "responseDecryptionIssuer" >CN=local-demo-cert,
OU=DIGIT, O=European Commission, L=Brussels, ST=Belgium, C=BE</ entry > < ent
ry   key = "serialNumber" >54C8F779</ entry >
    <!-- Key Agreement config -->
    <!-- If not present then no decryption will be applied on response -->
    <!--
    < entry   key = "responseDecryptionIssuer" >CN=local-ka-demo-cert,
OU=DIGIT, O=European Commission, L=Brussels, ST=Belgium, C=BE</ entry >
  < entry   key = "serialNumber" >2EAAA5F6A93C6BB0581DC8388501FAD06F77F394</
entry >
    -->
</ properties >
```

To enable the Key agreement using the provided keystores comment the entries under

```
<!-- Key transport config -->
```

and uncomment the entries under

```
<!-- Key Agreement config -->.
```

Afterwards, a server restart is advisable to enable the new configuration.

The private key described in the "responseDecryptionIssuer" entry is not actually the only key that will be used for the decryption.

This entry described the certificate that will be published in the metadata but all the private key entries from the keystore could be used to decrypt the SAML Response message.

Hence, the content of the keystore should always be as restrictive as it can be.

It is also possible to configure both the signature and the encryption with an HSM configuration, here is an example of HSM configuration

```
<? xml   version = "1.0"   encoding = "UTF-8" ?>
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
```

```
    <!-- Signature configuration with YubiHSM -->
    < entry   key = "keyStorePassword" >0001password</ entry >
    < entry   key = "keyStoreType" >PKCS11</ entry >
    < entry   key = "keyStoreProvider" >SunPKCS11-ykcs11</ entry >

    < entry   key = "issuer" >issuerDN</ entry >
    < entry   key = "serialNumber" >serialNumber</ entry >

    <!-- Metadata Signature configuration -->
    < entry   key = "metadata.keyStorePassword" >0001password</ entry >
    < entry   key = "metadata.keyStoreType" >PKCS11</ entry >
    < entry   key = "metadata.keyStoreProvider" >SunPKCS11-ykcs11</ entry >

    < entry   key = "metadata.issuer" >issuerDN</ entry >
    < entry   key = "metadata.serialNumber" >serialNumber</ entry >
</ properties >
```

Finally, the SAML engine instantiation should be like this:

```
ProtocolEngine engine =
ProtocolEngineFacotry.getInstance().getProtocolEngine ("CONF1");
```

If it is necessary, more instances could be created:

```
< instance   name = "CONF2" >
--------
</ instance >
ProtocolEngine engine2 =
ProtocolEngineFacotry.getInstance().getProtocolEngine ("CONF2");
< instance   name = "CONF_SPAIN" >
--------
</ instance >
ProtocolEngine engine =
ProtocolEngineFacotry.getInstance().getProtocolEngine ("CONF_SPAIN ");
```

5.4 Using eIDAS SAML Engine (public interfaces)

To generate the binary representation of the Request:

```
IRequestMessage generateRequestMessage(
```

```
     @Nonnull  IAuthenticationRequest request,
     @Nonnull  String serviceIssuer)
         throws  EIDASSAMLEngineException;
request: ready made EidasAuthenticationRequest
serviceIssuer: service provider URL (destination / Metadata URL)
```

To generate the binary representation of the Response:

```
IResponseMessage generateResponseMessage(
     @Nonnull  IAuthenticationRequest request,
     @Nonnull  IAuthenticationResponse response,
     boolean  signAssertion,
     @Nonnull  String ipAddress)  throws  EIDASSAMLEngineException;
request: the original request we create the response  for
response: ready made EidasAuthenticationResponse
signAssertion:  if  signature is needed
ipAddress: IP of client as seen in the national infrastructure (not used in
the Node - yet)
```

To deal with binary Request, unserialize it to *EidasAuthenticationRequest* object:

```
IAuthenticationRequest unmarshallRequestAndValidate(
     @Nonnull   byte [] requestBytes,
     @Nonnull  String citizenCountryCode)  throws  EIDASSAMLEngineException

requestBytes: the SAML message received
citizenCountryCode: county code of citizen to be authenticated (not part of
the standard EIDAS data
```

To deal with binary Response, unmarshall it to *EidasAuthenticationResponse* object:

```
Correlated unmarshallResponse(
     @Nonnull   byte [] responseBytes
)  throws  EIDASSAMLEngineException;

responseBytes: the SAML message received
```

Or:

```
IAuthenticationResponse unmarshallResponseAndValidate(
     @Nonnull   byte [] responseBytes,
     @Nonnull  String userIpAddress,
     long  beforeSkewTimeInMillis,
     long  afterSkewTimeInMillis,
     @Nullable  String audienceRestriction)  throws
 EIDASSAMLEngineException;

responseBytes: the SAML message received
userIpAddress: user IP address from web  interface  (not used in the Node –
yet)
beforeSkewTimeInMillis: allowed skew time
afterSkewTimeInMillis: allowed skew time
audienceRestriction: own Metadata URL (consumer address)
```

With the unmarshalled

```
IAuthenticationResponse validateUnmarshalledResponse(
     @Nonnull  Correlated unmarshalledResponse,
     @Nonnull  String userIpAddress,
     long  beforeSkewTimeInMillis,
     long  afterSkewTimeInMillis,
     @Nullable  String audienceRestriction)  throws
 EIDASSAMLEngineException;

unmarshalledResponse: the unmarshalled Response with correlating original
Request
userIpAddress: user IP address from web  interface  (not used in the Node –
yet)
beforeSkewTimeInMillis: allowed skew time
afterSkewTimeInMillis: allowed skew time
audienceRestriction: own Metadata URL (consumer address)
```

6 ProtocolEngine Configuration

6.1 Obtaining a ProtocolEngine instance

In eIDAS-Node (since version 1.1), the *ProtocolEngine* (*eu.eidas.auth.engine.ProtocolEngine*) replaces the deprecated *SAMLEngine*.

The protocol engine is responsible for implementing the protocol between the eIDAS-Node Connector and the eIDAS-Node Proxy Service. The default protocol engine strictly implements the eIDAS specification.

However ,with some effort, the protocol engine can be customised to implement protocols other than eIDAS. A *ProtocolEngine* instance is obtained from a *ProtocolEngineFactory* (*eu.eidas.auth.engine.ProtocolEngineFactory*).

There is a default *ProtocolEngineFactory*, *eu.eidas.auth.engine.DefaultProtocolEngineFactory* which uses the default configuration files.

You can obtain the protocol engine named " ***#MyEngineName#*** " by using the following statement:

```
ProtocolEngineI protocolEngine =
DefaultProtocolEngineFactory.getInstance().getProtocolEngine(
"#MyEngineName#" );
```

You can also achieve the same result using a convenient method in ProtocolEngineFactory via the getDefaultProtocolEngine method:

```
ProtocolEngineI engine =
ProtocolEngineFactory.getDefaultProtocolEngine( "#MyEngineName#" );
```

6.2 Configuring protocol engines

Protocol engines are created from a *ProtocolEngineConfiguration* (*eu.eidas.auth.engine.configuration.ProtocolEngineConfiguration*).

ProtocolEngineConfiguration instances are obtained from a *ProtocolEngineConfigurationFactory* (*eu.eidas.auth.engine.configuration.dom.ProtocolEngineConfigurationFactory*).

There is a default *ProtocolEngineConfigurationFactory*: (*eu.eidas.auth.engine.configuration.dom.DefaultProtocolEngineConfigurationFactory)* which uses the default configuration files.

DefaultProtocolEngineConfigurationFactory is the factory used by the *DefaultProtocolEngineFactory* to configure default protocol engine instances.

You can create your own *ProtocolEngineConfigurationFactory* and use it to create your own *ProtocolEngineFactory* which would not rely on the default configuration files.

For example the following is a Spring configuration snippet to create a custom *ProtocolEngineConfigurationFactory* and the corresponding *ProtocolEngineFactory*:

```
< bean   id = "NodeSamlEngineConfigurationFactory"   class = "eu.eidas.auth.
engine.configuration.dom.ProtocolEngineConfigurationFactory" >
    < constructor -arg  value = "SamlEngine.xml" />
    < constructor -arg  value = "#{eidasConfigFilePath}" />
    < constructor -arg  value = "#{eidasConfigRepository}" />
</ bean >

< bean   id = "NodeProtocolEngineFactory"   class = "eu.eidas.auth.engine.Pr
otocolEngineFactory" >
    < constructor -arg  ref = "NodeSamlEngineConfigurationFactory" />
</ bean >
```

6.2.1 The DefaultProtocolEngineConfigurationFactory

The default ProtocolEngineConfigurationFactory uses a configuration file called **SamlEngine.xml**.

The format of this file is as follows:

```
< instances >
     < instance   name = "MyEngineName" >

        < configuration   name = "SamlEngineConf" >
            < parameter   name = "fileConfiguration"   value = "SamlEngine_M
yEngineName.xml" />
        </ configuration >

        < configuration   name = "SignatureConf" >
            < parameter   name = "class"   value = "eu.eidas.auth.engine.cor
e.impl.SignSW" />
            < parameter   name = "fileConfiguration"   value = "SignModule_M
yEngineName.xml" />
        </ configuration >
        < configuration   name = "EncryptionConf" >
            <!-- Specific signature module -->
            < parameter   name = "class"   value = "eu.eidas.auth.engine.cor
e.impl.EncryptionSW" />
            <!-- Settings specific module responseTo/FromPointAlias &
requestTo/FromPointAlias parameters will be added -->
            < parameter   name = "fileConfiguration"   value = "EncryptModul
e_MyEngineName.xml" />

        </ configuration >

        <!-- Settings for the ExtensionProcessor module -->
```

```
        < configuration   name = "ProtocolProcessorConf" >
            <!-- Specific ExtensionProcessor module -->
            < parameter   name = "class"   value = "eu.eidas.auth.engine.cor
e.eidas.EidasProtocolProcessor"  />

            < parameter   name = "coreAttributeRegistryFile"
                        value = "saml-engine-eidas-attributes-
MyEngineName.xml"  />
            < parameter   name = "additionalAttributeRegistryFile"
                        value = "saml-engine-additional-attributes-
MyEngineName.xml"  />
        </ configuration >

        <!-- Settings for the Clock module -->
        < configuration   name = "ClockConf" >
            <!-- Specific Clock module -->
            < parameter   name = "class"
                        value = "eu.eidas.auth.engine.SamlEngineSystemClock"
 />
        </ configuration >

    </ instance >

</ instances >
```

The *SamlEngine.xml* file can be put on the *filesystem* or in the *classpath*. It is first looked for in the *classpath* but if not found, is loaded from the *filesystem*.

If the *SamlEngine.xml* file is available as a file URL (i.e. file://*somepath*), it is reloadable and will be reloaded as soon as the file is modified and its last-modified date attribute changes.

On the contrary, if the file is available from a jar, war or ear URL, it is not reloadable.

Therefore, if you want the ProtocolEngine configuration to be reloadable at runtime, put it in a folder in the *classpath* outside of an archive (jar, war, ear).

The *SamlEngine.xml* file contains a sequence of instances. An instance represents one configuration of a ProtocolEngine. A given ProtocolEngine is obtained by its name (in the example "*MyEngineName*") which must be unique per configuration file. In EIDAS, the SamlEngine.xml defines 2 instances called "Connector" and "Service". The Connector instance defines the ProtocolEngine used by the EIDAS-Connector while the Service instance defines the ProtocolEngine used by the EIDAS-ProxyService.

An instance is mapped to a *ProtocolEngineConfiguration*.

The *ProtocolEngineConfiguration* is composed of:

- a unique name (e.g. *MyEngineName*);
- core properties (which are configured via a configuration entry called *SamlEngineConf*);
- a Signature configuration entry (called *SignatureConf*);
- an (optional) encryption configuration entry (called *EncryptionConf*);
- a ProtocolProcessor configuration entry (called *ProtocolProcessorConf*);and
- a Clock configuration entry (called *ClockConf*).

6.2.2 Core properties

Protocol engine core properties are configured using the following configuration entry:

```
< configuration   name = "SamlEngineConf" >
    < parameter   name = "fileConfiguration"   value = "SamlEngine_MyEngineName.xml" />
</ configuration >
```

This entry is mapped to an implementation of the *eu.eidas.auth.engine.core.SamlEngineCoreProperties* interface.

Each core property can be configured directly inside the configuration entry (using parameters) or configured in an external file. The external file is referenced through a special parameter called "*fileConfiguration*". This file can be put on the *filesystem* or in the *classpath*. It is first looked for in the *classpath* but if not found, is loaded from the *filesystem*.

It is not necessary to use an external file, all core properties can be configured directly using parameters in the *SamlEngine.xml* file. If an external file is used, its format is a standard Java Properties file using either the .properties format or the .xml format.

If the Properties file is available as a file URL (i.e. file://*somepath*), it is reloadable and will be reloaded as soon as the file is modified and its last-modified date attribute changes.

On the contrary, if the file is available from a jar, war or ear URL, it is not reloadable.

Therefore if you want the core properties configuration to be reloadable at runtime, put it in a folder in the *classpath* outside of an archive (jar, war, ear).

The following is an example external file for core properties:

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
   < comment >SAML constants for AuthnRequests and Responses.</ comment >

   <!--
      Types of consent obtained from the user for this authentication and
      data transfer.
      Allow values: 'unspecified'.
   -->
   < entry   key = "consentAuthnRequest" >unspecified</ entry >

   <!--
   Allow values: 'obtained', 'prior', 'current-implicit', 'current-
explicit', 'unspecified'.
   -->
   < entry   key = "consentAuthnResponse" >obtained</ entry >

   <!--URI representing the classification of the identifier
      Allow values: 'entity'.
```

```
    -->
    < entry   key = "formatEntity" >entity</ entry >
    <!--Only HTTP-POST binding is supported for inter Node-->
    <!--The SOAP binding is only supported for direct communication between
SP-MW and VIdP-->
    < entry   key = "protocolBinding" >HTTP-POST</ entry >

    <!--eIDAS Node in the Service Provider's country-->
    < entry   key = "requester" > http://eIDASNode.gov.xx </ entry >

    <!-- eIDAS Node in the citizen's origin country-->
    < entry   key = "responder" > http://eIDASNode.gov.xx </ entry >

    <!--Subject cannot be confirmed on or after this number of seconds
(positive number)-->
    < entry   key = "timeNotOnOrAfter" >300</ entry >

    <!--Validation IP of the response-->
    < entry   key = "ipAddrValidation" >false</ entry >
    <!--allow unencrypted responses-->
    < entry   key = "allowUnencryptedResponse" >true</ entry >

</ properties >
```

The following is an example external file for the EIDAS-Connector core properties

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">

< properties >
    < comment >SAML constants for AuthnRequests.</ comment >

    <!--
       Types of consent obtained from the user for this authentication and
       data transfer.
       Allow values: 'unspecified'.
    -->
    < entry   key = "consentAuthnRequest" >unspecified</ entry >

    <!--URI representing the classification of the identifier
       Allow values: 'entity'.
    -->
    < entry   key = "formatEntity" >entity</ entry >

    <!--Only HTTP-POST binding is supported for inter Node-->
```

```
    <!--The SOAP binding is only supported for direct communication between
SP-MW and VIdP-->
    < entry   key = "protocolBinding" >HTTP-POST</ entry >

    <!--eIDAS Node in the Service Provider's country-->
    < entry   key = "requester" > http://eIDASNode.gov.xx </ entry >

    <!-- eIDAS Node in the citizen's origin country-->
    < entry   key = "responder" > http://eIDASNode.gov.xx </ entry >

    <!--Subject cannot be confirmed on or after this number of seconds
(positive number)-->
    < entry   key = "timeNotOnOrAfter" >300</ entry >

    <!--Validation IP of the response-->
    < entry   key = "ipAddrValidation" >false</ entry >
    <!--allow unencrypted responses-->
    < entry   key = "allowUnencryptedResponse" >true</ entry >

</ properties >
```

The following is an example external file for the EIDAS-ProxyService core properties

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">

< properties >
    < comment >SAML constants for AuthnResponses</ comment >

    <!--
    Allow values: 'obtained', 'prior', 'current-implicit', 'current-
explicit', 'unspecified'.
    -->
    < entry   key = "consentAuthnResponse" >obtained</ entry >

    <!--URI representing the classification of the identifier
       Allow values: 'entity'.
    -->
    < entry   key = "formatEntity" >entity</ entry >

    <!--Only HTTP-POST binding is supported for inter Node-->
    <!--The SOAP binding is only supported for direct communication between
SP-MW and VIdP-->
    < entry   key = "protocolBinding" >HTTP-POST</ entry >

    <!--eIDAS Node in the Service Provider's country-->
```

```
    < entry   key = "requester" > http://eIDASNode.gov.xx </ entry >

    <!-- eIDAS Node in the citizen's origin country-->
    < entry   key = "responder" > http://eIDASNode.gov.xx </ entry >

    <!--Subject cannot be confirmed on or after this number of seconds
(positive number)-->
    < entry   key = "timeNotOnOrAfter" >300</ entry >

    <!--Validation IP of the response-->
    < entry   key = "ipAddrValidation" >false</ entry >
    <!--allow unencrypted responses-->
    < entry   key = "allowUnencryptedResponse" >true</ entry >

</ properties >
```

The core property keys can be found in the *eu.eidas.auth.engine.core.SAMLCore* enum.

The most important core properties are:

- formatEntity
- ipAddrValidation
- oneTimeUse
- consentAuthnRequest (in EIDAS defined only by the Connector ProtocolEngine)
- timeNotOnOrAfter
- requester
- responder
- validateSignature
- consentAuthnResponse (in EIDAS defined only by the Service ProtocolEngine)
- protocolBinding
- messageFormat.eidas

There are also alternative properties such as :

Key	Description
enable.address.attribute.subject.confirmation.data	Enable or disable adding ipAddress to SubjectConfirmationData for SAML response assertion Default value is false (and therefore the addition of ipAddress is disabled by default)

The list of all core property keys and values is available in the *eIDAS-Node Installation, Configuration and Integration Manual*.

6.2.3 Signature Configuration

The following is an example of the signature configuration entry:

```
< configuration   name = "SignatureConf" >
```

```
    < parameter   name = "class"   value = "eu.eidas.auth.engine.core.impl.S
ignSW" />
    < parameter   name = "fileConfiguration"   value = "SignModule_MyEngineN
ame.xml" />
</ configuration >
```

It contains a class parameter which must have as value the name of a class available in the *classpath* which implements the *eu.eidas.auth.engine.core.ProtocolSignerI* interface.

A base abstract class to implement this interface can be *eu.eidas.auth.engine.core.impl.AbstractProtocolSigner*.

Note: The given class should also implement the *eu.eidas.auth.engine.metadata.MetadataSignerI* interface to sign eIDAS metadata documents.

The configured implementing class must have a public constructor taking as single argument a *java.util.Map of <String, String>*.

The signature configuration must also contain signature properties.
The signature properties correspond to the *eu.eidas.auth.engine.configuration.dom.SignatureConfiguration* class and all signature property keys are defined in the *eu.eidas.auth.engine.configuration.dom.SignatureKey* enum.

Each signature property can be configured directly inside the configuration entry (using parameters) or configured in an external file. The external file is referenced through a special parameter called "*fileConfiguration*". This file can be put on the filesystem or in the *classpath*. It is first looked for in the *classpath* but if not found, is loaded from the *filesystem*.

It is not necessary to use an external file, all signature properties can be configured by using parameters directly in the *SamlEngine.xml* file. If an external file is used, its format is a standard Java Properties file using either the .properties format or the .xml format. If the Properties file is available as a file URL (i.e. file://*somepath*).

The following is an example of an external file for signature properties:

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < comment >SWModule sign with JCEKS.</ comment >
    < entry   key = "check.certificate.validity.period" >true</ entry >
    < entry   key = "disallow.self.signed.certificate" >false</ entry >
    < entry   key = "keyStorePath" >MyKeyStore.jceks</ entry >
    < entry   key = "keyStorePassword" >local-demo</ entry >
    < entry   key = "keyPassword" >local-demo</ entry >
    < entry   key = "issuer" >CN=local-demo-cert, OU=DIGIT, O=European
Commission, L=Brussels, ST=Belgium, C=BE</ entry >
    < entry   key = "serialNumber" >54c8f779</ entry >
    < entry   key = "keyStoreType" >JCEKS</ entry >
</ properties >
```

The following table shows the various signature & encryption property keys.

Table 3: Signature & Encryption property keys

Signature keys	
Key	**Description**
request.sign.with.key.value	When set to true, the signature of the signed Authentication Requests will inclu When set to false the signature will include the full X509Certificate. Default value is false.
response.sign.with.key.value	When set to true, the signature of the signed Authentication Responses will incl When set to false the signature will include the full X509Certificate. Default value is false.
response.sign.assertions	When set to true, the SAML Responses (generated in Proxy Service) will have th Default value is false.
signature.algorithm	This configuration entry defines the signing algorithm (SHA2 based) that will be Possible values are: http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha384-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha512-rsa-MGF1 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512 The default configuration value is: http://www.w3.org/2007/05/xmldsig-more#ecdsa-sha512
metadata.signature.algorithm	This configuration entry defines the signing algorithm (SHA2 based) that will be Possible values are: http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha384-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha512-rsa-MGF1 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512 The default configuration value is: http://www.w3.org/2007/05/xmldsig-more#ecdsa-sha512

signature.algorithm.whitelist	The list of allowed signature algorithms. It contains OpenSAML's supported sig Currently the elements of the list s may be picked from the following (which is t http://www.w3.org/2007/05/xmldsig-more#sha256-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha384-rsa-MGF1 http://www.w3.org/2007/05/xmldsig-more#sha512-rsa-MGF1 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512
digest.method.algorithm	This allow to configure the digest method algorithm that will be used to sign. If not specified the default digest method algorithm will be : http://www.w3.org/2001/04/xmlenc#sha512
digest.method.algorithm.whitelist	This allow to configure the whitelist of digest method algorithms that will be us If not specify the following list of digest method algorithms will be used: • http://www.w3.org/2001/04/xmlenc#sha256 • http://www.w3.org/2001/04/xmldsig-more#sha384 • http://www.w3.org/2001/04/xmlenc#sha512
Encryption keys	
Key	**Description**
response.encryption.mandatory	Specifies whether encryption must always be used. When set to 'true' assertion responses. Default value is true. **Note**: this parameter is used by both Service and Connector.
data.encryption.algorithm	Specifies the data encryption algorithm (optional) (If not specified, the default
key.encryption.algorithm	Specifies the key encryption algorithm (optional) when Key Transport is used. T This property is only needed in the encryption configuration of the proxy servic
key.encryption.algorithm.key.wrapping	Specifies the key encryption algorithm (optional) when Key Agreement is used. This property is only needed in the encryption configuration of the proxy servic
key.encryption.agreement.method.algorithm	Specifies the key encryption agreement method algorithm (optional) when Key The default algorithm used is: http://www.w3.org/2009/xmlenc11#ECDH-ES in This property is only needed in the encryption configuration of the proxy servic

assertion.encrypt.with.key.value	When set to true, the eIDAS-Node marshals the encrypted assertions added to t This will only work with RSA keys. Default is false.
encryption.algorithm.whitelist	Specifies the white list of allowed data encryption algorithms (comma or semi- If not specified, the default white list is: http://www.w3.org/2009/xmlenc11#aes256-gcm; http://www.w3.org/2009/xmlenc11#aes192-gcm; http://www.w3.org/2009/xmlenc11#aes128-gcm; Those are the only three values allowed by the 1.2. technical specifications so n
jcaProviderName	Specifies the name of the Java Cryptography Architecture (JCA) Security Provid
responseDecryptionIssuer	Specifies the issuer DN of the certificate that will be published in the metadata Note**:** This will determine if key agreement or key transport is used based on th Note**:** All keys present in the keystore can be used for the decryption of the resp
responseToPointIssuer.<country code> e.g. responseToPointIssuer.BE	The issuer of the certificate to be used to encrypt responses to the given countr
responseToPointSerialNumber.<country code> e.g. responseToPointSerialNumber.BE	The serial number of a certificate to be used to encrypt responses to the given c
encryptionActivation:	Specifies the name of the encryption activation file to be loaded from the classp
Shared (Signature & Encryption) keys	
Key	**Description**
check.certificate.validity.period	Specifies whether certificate validity period is enforced (true) or whether expire Default value: true
disallow.self.signed.certificate	Specifies whether self-signed certificates cannot be used (this applies to signat Default value: false
serialNumber	Specifies the serialNumber of the certificate published in the metadata.
keyStorePath	Specifies the name of the keyStore to be loaded from the classpath or the path
keyStoreType	Specifies the type of the keyStore (optional) (If not specified, the default Java k

keyStoreProvider	Specifies the provider of the keyStore (optional) (If not specified, all the installe
keyStorePassword	Specifies the password of the keyStore.
keyPassword	Specifies the password of the private key used to perform decryptions.

6.2.4 The encryption activation file

Encryption can be activated country per country when the *response.encryption.mandatory* property is NOT set.

This is configured in the encryption activation file. The encryption activation file looks as follows:

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < entry   key = "EncryptTo.LU" >true</ entry >
    < entry   key = "DecryptFrom.LU" >true</ entry >

    < entry   key = "EncryptTo.FR" >true</ entry >
    < entry   key = "DecryptFrom.FR" >true</ entry >

    < entry   key = "EncryptTo.DE" >true</ entry >
    < entry   key = "DecryptFrom.DE" >true</ entry >

    < entry   key = "EncryptTo.BE" >false</ entry >
    < entry   key = "DecryptFrom.BE" >false</ entry >

</ properties >
```

If the *response.encryption.mandatory* property is set to "true", this file is ignored.

The various encryption property keys are:

- *EncryptTo + a valid country code* — whether encryption is turned on to the given country.
- *DecryptFrom + a valid country code* — whether decryption is turned on from the given country

If the *EncryptTo + a valid country code* property is not enabled, the Proxy Service does not encrypt the response sent to the Connector of the corresponding country.

The activation file properties can also be configured directly into the encryption file or protocol engine configuration file.

6.2.5 ProtocolProcessor configuration

The following is an example of the *ProtocolProcessor* configuration entry:

```
<!-- Settings for the ProtocolProcessor module -->
< configuration   name = "ProtocolProcessorConf" >
```

```
    <!-- Specific ExtensionProcessor module -->
    < parameter   name = "class"
                value =
"eu.eidas.auth.engine.core.eidas.EidasProtocolProcessor"  />

    < parameter   name = "coreAttributeRegistryFile"
                value = "protocol-engine-eidas-attributes.xml"  />

    < parameter   name = "additionalAttributeRegistryFile"
                value = "protocol-engine-additional-attributes.xml"  />

    < parameter   name = "metadataFetcherClass"
                value =
"eu.eidas.node.auth.metadata.SpringManagedMetadataFetcher" />
</ configuration >
```

It contains a *class* parameter which must have a value of the name of a class available in the *classpath* which implements the *eu.eidas.auth.engine.core.ProtocolProcessorI* interface.

An example of implementation for this interface can be *eu.eidas.auth.engine.core.eidas.EidasProtocolProcessor.*

The configured class implementing the *ProtocolProcessorI* interface is responsible for providing the actual protocol implementation.

eu.eidas.auth.engine.core.eidas.EidasProtocolProcessor provides the implementation of the eIDAS protocol.

Currently only SAML-based protocols are supported. The configured implementing class must have a **public** constructor with the following signature

```
public  SomeProtocolProcessor( @Nullable  AttributeRegistry
specAttributeRegistry,
                               @Nullable  AttributeRegistry
additionalAttributeRegistry,
                               @Nullable  MetadataFetcherI metadataFetcher,
                               @Nullable  MetadataSignerI metadataSigner)
{...
```

The first constructor argument is the attribute registry of the protocol specification (for example, the eIDAS minimum data sets).

The second constructor argument is the attribute registry of additional attributes (aka sector-specific attributes).

The third constructor argument is the implementation of the *MetadataFetcherI* interface when metadata are used (mandatory for the eIDAS protocol).

The fourth constructor argument is the implementation of the *MetadataSignerI* interface when metadata signing is enabled (example for the eIDAS protocol).

The *ProtocolProcessor* configuration can also contain properties.

The *ProtocolProcessor* property keys are defined in the *eu.eidas.auth.engine.configuration.dom.ParameterKey* enum.
The various *ProtocolProcessor* property keys are:

- *coreAttributeRegistryFile* the name of the core attribute registry file to be loaded from the classpath or the relative path of the core attribute registry file on the filesystem (optional).
- *additionalAttributeRegistryFile* the name of the additional attribute registry file to be loaded from the classpath or the relative path of the additional attribute registry file on the filesystem (optional).
- *metadataFetcherClass* the name of a class available in the classpath which implements the *eu.eidas.auth.engine.metadata.MetadataFetcherI* interface (optional).

The core *AttributeRegistry* is the attribute registry of the protocol specification (for example, the eIDAS minimum data sets).

The additional *AttributeRegistry* is the attribute registry of additional attributes (aka sector-specific attributes).

When no core attribute registry is configured, the registry of the eIDAS specification is used (*eu.eidas.auth.engine.core.eidas.spec.EidasSpec.REGISTRY*).

When no additional registry is configured, there is simply no additional attribute at all.

6.2.6 The Attribute Registry

The attribute registry contains the definitions of all the supported attributes. The attribute registry is implemented in the class *eu.eidas.auth.commons.attribute.AttributeRegistry*. An attribute registry can be instantiated programmatically with the *AttributeRegistry* class or loaded from a file.

This file can be put on the filesystem or in the classpath. It is first looked after in the classpath but if not found, it is loaded from the filesystem. Its format is a standard Java Properties file using either the .properties format or the .xml format.

If the Properties file is available as a file URL (i.e. file://somepath), it is reloadable and will be reloaded as soon as the file is modified and its last-modified date attribute changes. On the contrary, if the file is available from a jar, war or ear URL, it is not reloadable.

Therefore, if you want the attribute registry to be reloadable at runtime, put it in a folder in the classpath outside of an archive (jar, war, ear).

Example attribute registry complying with the eIDAS specification:

```
<!DOCTYPE properties SYSTEM " http://java.sun.com/dtd/properties.dtd ">
< properties >
    < comment >eIDAS attributes</ comment >
        < entry   key = "1.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/PersonIdentifier </ entry >
        < entry   key = "1.FriendlyName" >PersonIdentifier</ entry >
        < entry   key = "1.PersonType" >NaturalPerson</ entry >
        < entry   key = "1.Required" >true</ entry >
        < entry   key = "1.UniqueIdentifier" >true</ entry >
        < entry   key = "1.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "1.XmlType.LocalPart" >PersonIdentifierType</ entry >
        < entry   key = "1.XmlType.NamespacePrefix" >eidas-natural</ entry >
```

```
        < entry   key = "1.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "2.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/CurrentFamilyName </ entry >
        < entry   key = "2.FriendlyName" >FamilyName</ entry >
        < entry   key = "2.PersonType" >NaturalPerson</ entry >
        < entry   key = "2.Required" >true</ entry >
        < entry   key = "2.TransliterationMandatory" >true</ entry >
        < entry   key = "2.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "2.XmlType.LocalPart" >CurrentFamilyNameType</ entry >
        < entry   key = "2.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "2.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.StringAttributeValueMarshaller</ entry >
        < entry   key = "3.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/CurrentGivenName </ entry >
        < entry   key = "3.FriendlyName" >FirstName</ entry >
        < entry   key = "3.PersonType" >NaturalPerson</ entry >
        < entry   key = "3.Required" >true</ entry >
        < entry   key = "3.TransliterationMandatory" >true</ entry >
        < entry   key = "3.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "3.XmlType.LocalPart" >CurrentGivenNameType</ entry >
        < entry   key = "3.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "3.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.StringAttributeValueMarshaller</ entry >
        < entry   key = "4.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/DateOfBirth </ entry >
        < entry   key = "4.FriendlyName" >DateOfBirth</ entry >
        < entry   key = "4.PersonType" >NaturalPerson</ entry >
        < entry   key = "4.Required" >true</ entry >
        < entry   key = "4.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "4.XmlType.LocalPart" >DateOfBirthType</ entry >
        < entry   key = "4.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "4.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.DateTimeAttributeValueMarshaller</ entry >
        < entry   key = "5.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/BirthName </ entry >
        < entry   key = "5.FriendlyName" >BirthName</ entry >
        < entry   key = "5.PersonType" >NaturalPerson</ entry >
        < entry   key = "5.Required" >false</ entry >
        < entry   key = "5.TransliterationMandatory" >true</ entry >
        < entry   key = "5.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
```

```
        < entry   key = "5.XmlType.LocalPart" >BirthNameType</ entry >
        < entry   key = "5.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "5.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.StringAttributeValueMarshaller</ entry >
        < entry   key = "6.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/PlaceOfBirth </ entry >
        < entry   key = "6.FriendlyName" >PlaceOfBirth</ entry >
        < entry   key = "6.PersonType" >NaturalPerson</ entry >
        < entry   key = "6.Required" >false</ entry >
        < entry   key = "6.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "6.XmlType.LocalPart" >PlaceOfBirthType</ entry >
        < entry   key = "6.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "6.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "7.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/CurrentAddress </ entry >
        < entry   key = "7.FriendlyName" >CurrentAddress</ entry >
        < entry   key = "7.PersonType" >NaturalPerson</ entry >
        < entry   key = "7.Required" >false</ entry >
        < entry   key = "7.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "7.XmlType.LocalPart" >CurrentAddressType</ entry >
        < entry   key = "7.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "7.AttributeValueMarshaller" >eu.eidas.auth.commons.
protocol.eidas.impl.CurrentAddressAttributeValueMarshaller</ entry >
        < entry   key = "8.NameUri" > http://eidas.europa.eu/attributes/
naturalperson/Gender </ entry >
        < entry   key = "8.FriendlyName" >Gender</ entry >
        < entry   key = "8.PersonType" >NaturalPerson</ entry >
        < entry   key = "8.Required" >false</ entry >
        < entry   key = "8.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/naturalperson </ entry >
        < entry   key = "8.XmlType.LocalPart" >GenderType</ entry >
        < entry   key = "8.XmlType.NamespacePrefix" >eidas-natural</ entry >
        < entry   key = "8.AttributeValueMarshaller" >eu.eidas.auth.commons.
protocol.eidas.impl.GenderAttributeValueMarshaller</ entry >
        < entry   key = "9.NameUri" > http://eidas.europa.eu/attributes/
legalperson/LegalPersonIdentifier </ entry >
        < entry   key = "9.FriendlyName" >LegalPersonIdentifier</ entry >
        < entry   key = "9.PersonType" >LegalPerson</ entry >
        < entry   key = "9.Required" >true</ entry >
        < entry   key = "9.UniqueIdentifier" >true</ entry >
        < entry   key = "9.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
```

```
        < entry   key = "9.XmlType.LocalPart" >LegalPersonIdentifierType</
entry >
        < entry   key = "9.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "9.AttributeValueMarshaller" >eu.eidas.auth.commons.
attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "10.NameUri" > http://eidas.europa.eu/attributes/
legalperson/LegalName </ entry >
        < entry   key = "10.FriendlyName" >LegalName</ entry >
        < entry   key = "10.PersonType" >LegalPerson</ entry >
        < entry   key = "10.Required" >true</ entry >
        < entry   key = "10.TransliterationMandatory" >true</ entry >
        < entry   key = "10.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "10.XmlType.LocalPart" >LegalNameType</ entry >
        < entry   key = "10.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "10.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.StringAttributeValueMarshaller</ entry >
        < entry   key = "11.NameUri" >  http://eidas.europa.eu/attributes/
legalperson/LegalPersonAddress </ entry >
        < entry   key = "11.FriendlyName" >LegalAddress</ entry >
        < entry   key = "11.PersonType" >LegalPerson</ entry >
        < entry   key = "11.Required" >false</ entry >
        < entry   key = "11.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "11.XmlType.LocalPart" >LegalPersonAddressType</
entry >
        < entry   key = "11.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "11.AttributeValueMarshaller" >eu.eidas.auth.commons
.protocol.eidas.impl.LegalAddressAttributeValueMarshaller</ entry >
        < entry   key = "12.NameUri" >  http://eidas.europa.eu/attributes/
legalperson/VATRegistrationNumber </ entry >
        < entry   key = "12.FriendlyName" >VATRegistration</ entry >
        < entry   key = "12.PersonType" >LegalPerson</ entry >
        < entry   key = "12.Required" >false</ entry >
        < entry   key = "12.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "12.XmlType.LocalPart" >VATRegistrationNumberType</
entry >
        < entry   key = "12.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "12.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "13.NameUri" > http://eidas.europa.eu/attributes/
legalperson/TaxReference </ entry >
        < entry   key = "13.FriendlyName" >TaxReference</ entry >
        < entry   key = "13.PersonType" >LegalPerson</ entry >
```

```
        < entry   key = "13.Required" >false</ entry >
        < entry   key = "13.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "13.XmlType.LocalPart" >TaxReferenceType</ entry >
        < entry   key = "13.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "13.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "14.NameUri" > http://eidas.europa.eu/attributes/
legalperson/D-2012-17-EUIdentifier </ entry >
        < entry   key = "14.FriendlyName" >D-2012-17-EUIdentifier</ entry >
        < entry   key = "14.PersonType" >LegalPerson</ entry >
        < entry   key = "14.Required" >false</ entry >
        < entry   key = "14.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "14.XmlType.LocalPart" >D-2012-17-EUIdentifierType</
entry >
        < entry   key = "14.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "14.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "15.NameUri" > http://eidas.europa.eu/attributes/
legalperson/LEI </ entry >
        < entry   key = "15.FriendlyName" >LEI</ entry >
        < entry   key = "15.PersonType" >LegalPerson</ entry >
        < entry   key = "15.Required" >false</ entry >
        < entry   key = "15.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "15.XmlType.LocalPart" >LEIType</ entry >
        < entry   key = "15.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "15.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "16.NameUri" > http://eidas.europa.eu/attributes/
legalperson/EORI </ entry >
        < entry   key = "16.FriendlyName" >EORI</ entry >
        < entry   key = "16.PersonType" >LegalPerson</ entry >
        < entry   key = "16.Required" >false</ entry >
        < entry   key = "16.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "16.XmlType.LocalPart" >EORIType</ entry >
        < entry   key = "16.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "16.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "17.NameUri" > http://eidas.europa.eu/attributes/
legalperson/SEED </ entry >
        < entry   key = "17.FriendlyName" >SEED</ entry >
        < entry   key = "17.PersonType" >LegalPerson</ entry >
```

```
        < entry   key = "17.Required" >false</ entry >
        < entry   key = "17.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "17.XmlType.LocalPart" >SEEDType</ entry >
        < entry   key = "17.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "17.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
        < entry   key = "18.NameUri" > http://eidas.europa.eu/attributes/
legalperson/SIC </ entry >
        < entry   key = "18.FriendlyName" >SIC</ entry >
        < entry   key = "18.PersonType" >LegalPerson</ entry >
        < entry   key = "18.Required" >false</ entry >
        < entry   key = "18.XmlType.NamespaceUri" > http://eidas.europa.eu/
attributes/legalperson </ entry >
        < entry   key = "18.XmlType.LocalPart" >SICType</ entry >
        < entry   key = "18.XmlType.NamespacePrefix" >eidas-legal</ entry >
        < entry   key = "18.AttributeValueMarshaller" >eu.eidas.auth.commons
.attribute.impl.LiteralStringAttributeValueMarshaller</ entry >
</ properties >
```

In practice, such a file is not needed as it duplicates the *eu.eidas.auth.engine.core.eidas.spec.EidasSpec.REGISTRY*.

However, the above example can be modified to tweak the eIDAS specification support.

6.2.7 The attribute registry file format

The attribute registry file is composed of attribute definitions. They represent the *eu.eidas.auth.commons.attribute.AttributeDefinition* class. An attribute definition is composed of the following properties:

- *NameUri*: [mandatory]: the name URI of the attribute (full name and must be a valid URI).
- *FriendlyName*: [mandatory]: the friendly name of the attribute (short name).
- *PersonType*: [mandatory]: either NaturalPerson or LegalPerson.
- *Required*: [optional]: whether the attribute is required by the specification (and is part of the minimal data set which must be requested).
- *TransliterationMandatory*: [optional]: whether the attribute values must be transliterated if provided in non LatinScript variants. (This is mandatory for some eIDAS attributes).
- *UniqueIdentifier*: [optional]: whether the attribute is a unique identifier of the person (at least one unique identifier attribute must be present in authentication responses).
- *XmlType.NamespaceUri:* [mandatory]: the XML namespace URI for the attribute values, for example: http://www.w3.org/2001/XMLSchema for an XML Schema string.
- *XmlType.LocalPart:* [mandatory]: the name of the XML type for the attributes values, for example: string for an XML Schema string.
- *XmlType.NamespacePrefix:* [mandatory]: the name of the XML namespace prefix for the attributes values, for example: xs for an XML Schema string.
- *AttributeValueMarshaller:* [mandatory]: the name of a class available in the classpath which implements the *eu.eidas.auth.commons.attribute.AttributeValueMarshaller* interface.

Each attribute definition in the property file is assigned a unique id followed by a dot '.' which allows the parser to associate properties to one given attribute definition.

The unique id can be any string not containing a dot '.'.

A convention can be to use numbers as unique ids as in the example above.

All properties used by the parser are be found in *eu.eidas.auth.commons.attribute.AttributeSetPropertiesConverter.Suffix*.

The *eu.eidas.auth.commons.attribute.AttributeValueMarshaller* interface is responsible for converting the string representation of an attribute value into a Java type and vice versa.

6.2.8 Clock configuration

The *Clock* configuration entry looks as follows:

```
<!-- Settings for the Clock module -->
< configuration   name = "ClockConf" >
    <!-- Specific Clock module -->
    < parameter   name = "class"   value = "eu.eidas.auth.engine.SamlEngineS
ystemClock"  />
</ configuration >
```

It contains a *class* parameter which must have as value the name of a class available in the classpath which implements the *eu.eidas.auth.engine.SamlEngineClock* interface.

The configured implementing class must have a *public empty* constructor.

The clock interface is responsible for obtaining the system time.

6.2.9 Default SAML Engine configuration

The SAML Engine module has a default configuration file defaultSamlEngineProperties.xml, which holds the default configurations for this module. Those can be overridden by explicitly setting them on the external configuration files of the SAML Engine which are defined in SamlEngine.xml or in associated configuration files.

The eIDAS-Node external configuration files, such as eidas.xml, cannot be used to redefine and any configuration property will not be taken into account by the ProtocolEngine.

7 References

- Security Assertion Markup Language (SAML) v2.0 Technical Overview http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html;
- *eIDAS CryptoeIDAS*: Security Requirements for TLS and SAML;
- *eIDAS Interop*: Interoperability Architecture;
- *RFC5280IETF: RFC 5280*: D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk: RFC 5280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile;
- *SAML-CoreOASIS*: Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0;
- *SAML-BindingOASIS*: Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0;
- *SAML-SecOASIS*: F. Hirsch, R. Philpott, E. Maler: Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0;
- *SAML-MetaOASIS*: Metadata for the OASIS Security Assertion Markup Language (SAML) v2.0;
- *SAML-Meta: OASIS*: Metadata for the OASIS Security Assertion Markup Language (SAML) v2.0 http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf;
- *SAML-MetaIOP*: OASIS: SAML V2.0 Metadata Interoperability Profile Version 1.0 https://www.oasis-open.org/committees/download.php/36645/draft-sstc-metadata-iop-2.0-01.pdf;
- *SAML-MetaIOPOASIS*: SAML V2.0 Metadata Interoperability Profile Version 1.0;
- *XMLSig BPW3C*: XML Signature Best Practices, http://www.w3.org/TR/xmldsig-bestpractices.