Digital Signature Service Protocol Specifications

Version 0.9.0

Frank Cornelis

Published Dec 14, 2013

Abstract

This document details on the Digital Signature Service Protocol specifications.

1. Introduction 1
 1.1. Namespaces 2
 1.2. References 3
2. The basic protocol run 4
 2.1. Document uploading 4
 2.2. Browser POST 8
 2.3. Downloading the signed document 11
3. Extensions 14
 3.1. Signer Identity 14
 3.2. Localization 14
 3.3. SOAP with Attachments 14
4. Signature Verification 15
 4.1. Errors 17
5. Long-term validity of signatures 17
6. XML Schema 20
A. Digital Signature Service Protocol Specifications License 21
B. Digital Signature Service Protocol Project License 21

1. Introduction

The eID DSS 1.0.x product already defines a protocol between web applications and the DSS service. Although web application developers can secure this protocol, the protocol can be integrated in an insecure manner. This new Digital Signature Service Protocol features inherent security. This means that developers, when integrating the protocol, are forces to operate in a secure way by design. Thus the design of this new protocol features certain security properties that integrating parties no longer can circumvent.

We do not give a formal specification of the Digital Signature Service Protocol. Instead we define the protocol by means of a protocol run example. Instead of focusing on a formal specification, we find it more valuable to explain the choices we made when designing this protocol.

The protocol involves three parties:

- The web application that wants the end-user to sign a certain document.
- The DSS service that aids in letting the end-user to sign the document.
- The end-user (web browser client) that actually performs the document signing.

The signing protocols of the OASIS DSS specification mainly focus on centralized key management systems. Such architecture makes sense for situations where the connecting clients do not own tokens with signing capabilities themselves. However, large-scale signing token deployments (e.g. national eID cards) reduce the need for a centralized key management system. In such scenarios it is still interesting to keep a centralized system in place for several reasons:

- Despite the fact that every person owns a token with signing capability, he/she might not have the appropriate software installed on the system for the creation of electronic signatures. It might be easier to maintain a lightweight applet solution, instead of a full blown token middleware that has to be installed on every participating client's system. The diversity among the client platforms is also easier to manage from a centralized platform instead of by distributing token middleware to all participating clients. Furthermore, managing the configuration of the signature policy to be used for creation and validation of signatures within a certain business context might be easier using a centralized platform.
- When transforming a paper-world business workflow to a digital equivalent that includes the creation and/or validation of signatures, it might be interesting to offer the sub-process of creating/validating electronic signatures as an online service. Given the technicality of signature creation and validation, a clean separation of concerns in the service architecture is desired.
- From a technical point of view it might be easier to maintain different DSS servers each specializing in handling a specific token type. E.g. tokens per vendor, or per country.

So the role of the centralized system shifts from key management to providing a platform that manages the technicalities of signing documents using client tokens. Such digital signature service systems require a new set of protocol messages for the creation of signatures where signature computation is accomplished via local tokens.

1.1. Namespaces

The XML namespaces used in the following sections are described in *Table 1, “XML Namespaces”* .

Table 1. XML Namespaces

Prefix	Namespace
`dss`	`urn:oasis:names:tc:dss:1.0:core:schema`
`async`	`urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing:1.0`

Prefix	Namespace
ds	`http://www.w3.org/2000/09/xmldsig#`
soap	`http://www.w3.org/2003/05/soap-envelope`
wsa	`http://www.w3.org/2005/08/addressing`
wsse	`http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd`
wsu	`http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd`
wst	`http://docs.oasis-open.org/ws-sx/ws-trust/200512`
ec	`http://www.w3.org/2001/10/xml-exc-c14n#`
dssp	`urn:be:e-contract:dssp:1.0`
vr	`urn:oasis:names:tc:dss-x:1.0:profiles:verificationreport:schema#`

1.2. References

The following specifications are references:

[Base64] S. Josefsson, The Base16, Base32, and Base64 Data Encodings , The Internet Society, 2006 *http://tools.ietf.org/html/rfc4648* *[http://tools.ietf.org/html/rfc4648]*

[DSSAsync] A. Kuehne et al., Asynchronous Processing Abstract Profile of the OASIS Digital Signature Services Version 1.0 , OASIS, April 2007 *http://docs.oasis-open.org/dss/v1.0/oasis-dss-profiles-asynchronous_processing-spec-v1.0-os.pdf* *[http://docs.oasis-open.org/dss/v1.0/oasis-dss-profiles-asynchronous_processing-spec-v1.0-os.pdf]*

[DSSCore] S. Drees et al., Digital Signature Service Core Protocols and Elements , OASIS, April 2007 *http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.pdf* *[http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.pdf]*

[DSSVer] D. Hhnlein et al., Profile for Comprehensive Multi-Signature Verification Reports Version 1.0 , OASIS, November 2010 *http://docs.oasis-open.org/dss-x/profiles/verificationreport/oasis-dssx-1.0-profiles-vr-cs01.pdf* *[http://docs.oasis-open.org/dss-x/profiles/verificationreport/oasis-dssx-1.0-profiles-vr-cs01.pdf]*

[Excl-C14N] J. Boyer et al., Exclusive XML Canonicalization Version 1.0 , World Wide Web Consortium, July 2002 *http://www.w3.org/TR/xml-exc-c14n/* *[http://www.w3.org/TR/xml-exc-c14n/]*

[HTML401] D. Raggett et al., HTML 4.01 Specification , World Wide Web Consortium, December 1999 *http://www.w3.org/TR/html4* *[http://www.w3.org/TR/html4]*

[RFC 2616] R. Fielding et al., Hypertext Transfer Protocol - HTTP/1.1. , *http://www.ietf.org/rfc/rfc2616.txt* IETF (Internet Engineering Task Force) RFC 2616, June 1999.

[SOAP] W3C, SOAP Version 1.2 , *SOAP Version 1.2* *[http://www.w3.org/TR/soap12-part1/]* W3C Recommendation 27 April 2007

[SwA] WS-I, Attachments Profile Version 1.0 , *WS-I Attachments Profile Version 1.0* *[http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html]* Web Services Interoperability Organization

[RFC 2246] T. Dierks, C. Allen, The TLS Protocol Version 1.0 , *http://www.ietf.org/rfc/rfc2246.txt* IETF (Internet Engineering Task Force) RFC 2246, January 1999.

[WS-SecConv] A. Nadalin et al., WS-SecureConversation 1.4 , OASIS, February 2009 *http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.html* *[http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.html]*

[WS-Sec] Kelvin Lawrence, Chris Kaler, OASIS Web Services Security: SOAP Message Security 1.1 , OASIS, February 2006 *OASIS WS-Security 1.1* *[https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf]*

[WS-Trust] A. Nadalin et al., WS-Trust 1.3 , OASIS, March 2007 *http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html* *[http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html]*

[XHTML] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition) , World Wide Web Consortium Recommendation, August 2002 *http://www.w3.org/TR/xhtml1/* *[http://www.w3.org/TR/xhtml1/]*

[XMLSig] D. Eastlake et al., XML-Signature Syntax and Processing , W3C Recommendation, June 2008 *http://www.w3.org/TR/xmldsig-core/*

[XML-ns] T. Bray, D. Hollander, A. Layman, Namespaces in XML , W3C Recommendation, January 1999 *http://www.w3.org/TR/1999/REC-xml-names-19990114* *[http://www.w3.org/TR/1999/REC-xml-names-19990114]*

2. The basic protocol run

The basic protocol run consists of three request/response messages.

- The web application uploads the document to be signed to the DSS.
- Actual signing request from the web application towards the DSS.
- The web application downloads the signed document from the DSS.

2.1. Document uploading

A protocol run starts with a SOAP version 1.2 request *[SOAP]* from the web application towards the DSS. This first step allows the web application to send over the document to be signed to the DSS. It also allows for the web application and the DSS to establish a security context. The document is not transmitted via a Browser POST. Given the upload limitation of most end-user's internet connection, this might result in a bad end-user experience when trying to sign a large document.

```
<soap:Envelope>
    <soap:Body>
        <dss:SignRequest Profile="urn:be:e-contract:dssp:1.0">
            <dss:OptionalInputs>
                <dss:AdditionalProfile>
            urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing
                </dss:AdditionalProfile>
                <wst:RequestSecurityToken>
                    <wst:TokenType>
        http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
                    </wst:TokenType>
                    <wst:RequestType>
                http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
                    </wst:RequestType>
                    <wst:Entropy>
                        <wst:BinarySecret Type=
            "http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
                            ...
                        </wst:BinarySecret>
                    </wst:Entropy>
                    <wst:KeySize>256</wst:KeySize>
                </wst:RequestSecurityToken>
                <dss:SignaturePlacement WhichDocument="doc1"
                    CreateEnvelopedSignature="true"/>
                <dss:SignatureType>
                    urn:be:e_contract:dssp:signature:xades-x-l
                </dss:SignatureType>
            </dss:OptionalInputs>
            <dss:InputDocuments>
                <dss:Document ID="doc1">
                   <dss:Base64Data MimeType="...">the document</dss:Base64Data>
                </dss:Document>
            </dss:InputDocuments>
        </dss:SignRequest>
    </soap:Body>
</soap:Envelope>
```

We use the OASIS Asynchronous Abstract Profile *[DSSAsync]* given the nature of this first request.

Given the fact that the transmitted document can contain sensitive data, the SOAP request is transmitted over SSL *[RFC 2246]* . This allows the web application to authenticate and trust the DSS endpoint.

Via WS-SecureConversation *[WS-SecConv]* , the web application and DSS can establish a shared session key. This session key will be used in subsequent message exchanges. We do not use the WS-Addressing features as defined within the WS-SecureConversation specs, as we do not need these routing capabilities.

If required, the web application can authenticate itself by means of a WS-Security *[WS-Sec]* SOAP header. This allows the DSS to operate in two modes:

- Development mode, where the web applications can access the DSS unauthorized.
- Production mode, where the web applications must be authorized by the DSS.

Since the eID DSS 2.0 supports multiple signature types, we include an optional `<dss:SignatureType>` element. We define the following signature type URIs:

- `urn:be:e-contract:dssp:signature:xades-x-l`

  This signature type is compatible with the signatures created by eID DSS 1.0.x.
- `urn:be:e-contract:dssp:signature:xades-baseline`

  ETSI XAdES Baseline profile signatures.
- `urn:be:e-contract:dssp:signature:pades-baseline`

  ETSI PAdES Baseline profile signatures.

If the `<dss:SignatureType>` element is not provided, the DSS will determine the most appropriate signature type itself.

The DSS Server responds as follows:

```
<soap:Envelope>
    <soap:Body>
        <dss:SignResponse Profile="urn:be:e-contract:dssp:1.0">
            <dss:Result>
                <dss:ResultMajor>
urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing:resultmajor:Pending
                </dss:ResultMajor>
            </dss:Result>
            <dss:OptionalOutputs>
                <async:ResponseID>responseId</async:ResponseID>
                <wst:RequestSecurityTokenResponseCollection >
                    <wst:RequestSecurityTokenResponse>
                        <wst:TokenType>
        http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
                        </wst:TokenType>
                        <wst:RequestedSecurityToken>
                            <wsc:SecurityContextToken wsu:Id="token-ref">
                                <wsc:Identifier>
```

```
                                        token-id
                                    </wsc:Identifier>
                                </wsc:SecurityContextToken>
                            </wst:RequestedSecurityToken>
                            <wst:RequestedAttachedReference>
                                <wsse:SecurityTokenReference>
                                    <wsse:Reference URI="#token-ref"/>
                                </wsse:SecurityTokenReference>
                            </wst:RequestedAttachedReference>
                            <wst:RequestedUnattachedReference>
                                <wsse:SecurityTokenReference>
                                    <wsse:Reference
ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"
                URI="token-id" />
                                </wsse:SecurityTokenReference>
                            </wst:RequestedUnattachedReference>
                            <wst:RequestedProofToken>
                                <wst:ComputedKey>
            http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/PSHA1
                                </wst:ComputedKey>
                            </wst:RequestedProofToken>
                            <wst:Entropy>
                                <wst:BinarySecret Type=
            "http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
                                    ...
                                </wst:BinarySecret>
                            </wst:Entropy>
                            <wst:KeySize>256</wst:KeySize>
                            <wst:Lifetime>
                                <wsu:Created>...</wsu:Created>
                                <wsu:Expires>...</wsu:Expires>
                            </wst:Lifetime>
                        </wst:RequestSecurityTokenResponse>
                    </wst:RequestSecurityTokenResponseCollection>
                </dss:OptionalOutputs>
            </dss:SignResponse>
        </soap:Body>
</soap:Envelope>
```

Via the `<async:ResponseID>` element the web application can further reference the uploaded document that has to be signed. The secure conversation token and corresponding proof-of-possession key is to be used to secure the subsequent messages between application and DSS Server. The proof-of-possession key is generated using the `P_SHA-1` algorithm *[RFC 2246]* .

2.1.1. Errors

In case the DSS received an unsupported document format, the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of `urn:be:e-contract:dssp:1.0:resultminor:UnsupportedMimeType`

In case the DSS received an unsupported `<dss:SignatureType>` value, the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

`urn:be:e-contract:dssp:1.0:resultminor:UnsupportedSignatureType`

In case the DSS received a `<dss:SignatureType>` value that is not supported for the given mime type, the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

`urn:be:e-contract:dssp:1.0:resultminor:IncorrectSignatureType`

2.2. Browser POST

The next request/response messages use a Browser POST as the DSS requires the end-user web browser to interact with the end-user and signature creation device (e.g., a smart card).

The web application initiates a Browser POST towards the DSS Server by means of the following HTML page *[HTML401]* :

```
<html>
    <head><title>DSS Browser POST</title></head>
    <body>
        <p>Redirecting to the DSS Server...</p>
        <form name="BrowserPostForm" method="post"
            action="https://www.e-contract.be/dss-ws/start">
            <input type="hidden" name="PendingRequest" value="..."/>
        </form>
        <script type="text/javascript">
            window.onload = function() {
                document.forms["BrowserPostForm"].submit();
            };
        </script>
    </body>
</html>
```

Here the `PendingRequest` field of the HTML form contains the base64 encoded *[Base64]* `<async:PendingRequest>` message. This `<async:PendingRequest>` message looks as follows:

```
<async:PendingRequest Profile="urn:be:e-contract:dssp:1.0">
    <dss:OptionalInputs>
        <dss:AdditionalProfile>
            urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing
        </dss:AdditionalProfile>
        <async:ResponseID>responseId</async:ResponseID>
```

```
            <wsa:MessageID>
                uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
            </wsa:MessageID>
            <wsu:Timestamp>
                <wsu:Created>2001-09-13T08:42:00Z</wsu:Created>
                <wsu:Expires>2001-10-13T09:00:00Z</wsu:Expires>
            </wsu:Timestamp>
            <wsa:ReplyTo>
                <wsa:Address>web application landing page URL</wsa:Address>
            </wsa:ReplyTo>
            <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#hmac-sha1"/>
                    <ds:Reference URI="">
                        <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>...</ds:DigestValue/>
                    </ds:Reference>
                </ds:SignedInfo>
                <ds:SignatureValue>...</ds:SignatureValue>
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
                        <wsse:Reference ValueType=
            "http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"
                            URI="token-id" />
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
            </ds:Signature>
        </dss:OptionalInputs>
    </async:PendingRequest>
```

Via the `<async:ResponseID>` the web application references the document that was previously transmitted to the DSS Server via the SOAP call described under *Section 2.1, “Document uploading”* . The different WS-Addressing and XML signature elements are used to secure the transmitted message. The message-level signature is using the security token's corresponding proof-of-possession key. The XML signature is using `URI=""` to reference the document as the top-level `<async:PendingRequest>` element does not define an identifier attribute.

After signing the document (in the case of eID DSS version 2.0, the actual signing between DSS Server and signature creation device is using a proprietary protocol) the DSS Server responds with the following HTML page:

```
<html>
    <head><title>DSS Browser POST</title></head>
    <body>
        <p>Redirecting to the web application...</p>
        <form name="BrowserPostForm" method="post"
            action="value of PendingRequestContext Destination attribute">
            <input type="hidden" name="SignResponse" value="..."/>
        </form>
        <script type="text/javascript">
            window.onload = function() {
                document.forms["BrowserPostForm"].submit();
            };
        </script>
    </body>
</html>
```

Where the `SignResponse` field of the HTML form contains the base64 encoded `<dss:SignResponse>` message. This `<dss:SignResponse>` message looks as follows:

```
<dss:SignResponse Profile="urn:be:e-contract:dssp:1.0">
    <dss:Result>
        <dss:ResultMajor>
urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing:resultmajor:Pending
        </dss:ResultMajor>
    </dss:Result>
    <dss:OptionalOutputs>
        <async:ResponseID>responseId</async:ResponseID>
        <wsa:RelatesTo>previous MessageID value</wsa:RelatesTo>
        <wsu:Timestamp>
            <wsu:Created>2001-09-13T08:42:00Z</wsu:Created>
            <wsu:Expires>2001-10-13T09:00:00Z</wsu:Expires>
        </wsu:Timestamp>
        <wsa:To>web application landing page URL</wsa:To>
        <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#hmac-sha1"/>
                <ds:Reference URI="">
                    <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>...</ds:DigestValue/>
                </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>...</ds:SignatureValue>
            <ds:KeyInfo>
                <wsse:SecurityTokenReference>
                    <wsse:Reference ValueType=
    "http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"
                        URI="token-id" />
                </wsse:SecurityTokenReference>
            </ds:KeyInfo>
        </ds:Signature>
    </dss:OptionalOutputs>
</dss:SignResponse>
```

The DSS Server signs the message with the secure conversation token's proof-of-possession key. The web application should of course check this signature. However, compared to the eID DSS 1.0.x protocol, the consequences of a web application not checking this signature are less exploitable as no vital information is passed as part of this response message.

2.2.1. Errors

In case the end-user cancelled the signing operation, the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

```
urn:be:e-contract:dssp:1.0:resultminor:user-cancelled
```

In case the DSS service detects a problem with the client runtime environment, the service returns in `<SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

```
urn:be:e-contract:dssp:1.0:resultminor:client-runtime
```

2.3. Downloading the signed document

Finally the web application can request the signed document from the DSS Server via a SOAP call:

```
<soap:Envelope>
    <soap:Header>
        <wsse:Security soap:mustUnderstand="1">
            <wsu:Timestamp wsu:Id="timestamp">
                <wsu:Created>...</wsu:Created>
                <wsu:Expires>...</wsu:Expires>
            </wsu:Timestamp>
            <wsc:SecurityContextToken wsu:Id="token-ref">
                    <wsc:Identifier>
                        token-id
                    </wsc:Identifier>
            </wsc:SecurityContextToken>
            <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#hmac-sha1"/>
                    <ds:Reference URI="#timestamp">
                        <ds:Transforms>
                            <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>...</ds:DigestValue/>
                    </ds:Reference>
                    <ds:Reference URI="#body">
                        <ds:Transforms>
                            <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>...</ds:DigestValue/>
                    </ds:Reference>
                </ds:SignedInfo>
                <ds:SignatureValue>...</ds:SignatureValue>
                <ds:KeyInfo>
                    <wsse:SecurityTokenReference>
                        <wsse:Reference URI="#token-ref" />
                    </wsse:SecurityTokenReference>
                </ds:KeyInfo>
            </ds:Signature>
        </wsse:Security>
    </soap:Header>
    <soap:Body wsu:Id="body">
        <async:PendingRequest Profile="urn:be:e-contract:dssp:1.0">
            <dss:OptionalInputs>
                <dss:AdditionalProfile>
            urn:oasis:names:tc:dss:1.0:profiles:asynchronousprocessing
                </dss:AdditionalProfile>
                <async:ResponseID>responseId</async:ResponseID>
                <wst:RequestSecurityToken>
                    <wst:RequestType>
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/Cancel
                    </wst:RequestType>
                    <wst:CancelTarget>
                        <wsse:SecurityTokenReference>
                            <wsse:Reference ValueType=
        "http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"
                                URI="token-id"/>
                        </wsse:SecurityTokenReference>
                    </wst:CancelTarget>
                </wst:RequestSecurityToken>
            </dss:OptionalInputs>
        </async:PendingRequest>
    </soap:Body>
</soap:Envelope>
```

The web application has to sign the SOAP request using WS-Security to give the DSS the assurance that no other party downloads the signed document except the original web application.

The web application can also instantly cancel the security token by piggybacking a `<wst:RequestSecurityToken>` element.

And the DSS Server returns the signed document:

```
<soap:Envelope>
    <soap:Body>
        <dss:SignResponse Profile="urn:be:e-contract:dssp:1.0">
            <dss:Result>
                <dss:ResultMajor>
                    urn:oasis:names:tc:dss:1.0:resultmajor:Success
                </dss:ResultMajor>
                <dss:ResultMinor>
    urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:OnAllDocuments
                </dss:ResultMinor>
            </dss:Result>
            <dss:OptionalOutputs>
                <dss:DocumentWithSignature>
                    <dss:Document ID="doc1">
                        <dss:Base64Data MimeType="...">
                            the signed document
                        </dss:Base64Data>
                    </dss:Document>
                </dss:DocumentWithSignature>
                <wst:RequestSecurityTokenResponseCollection>
                    <wst:RequestSecurityTokenResponse>
                        <wst:RequestedTokenCancelled/>
                    </wst:RequestSecurityTokenResponse>
                </wst:RequestSecurityTokenResponseCollection>
            </dss:OptionalOutputs>
            <dss:SignatureObject>
                <dss:SignaturePtr WhichDocument="doc1"/>
            </dss:SignatureObject>
        </dss:SignResponse>
    </soap:Body>
</soap:Envelope>
```

3. Extensions

We define several extensions on the basic protocol run.

3.1. Signer Identity

The web application cannot always determine in advance which digital identity will be used by the end-user to sign the document. Different strategies are possible to eventually determine this digital identity. The web application can for example verify the signed document via a DSS verifying protocol as defined under section 4 of *[DSSCore]* for this. Of course, if multiple signature are present on the document, this can cause problems. Via the elements defined within this section we give the web application additional means to determine the signatory's digital identity.

We allow usage of the `<dss:ReturnSignerIdentity>` optional input and corresponding `<dss:SignerIdentity>` optional output as defined under section 4.5.7 of *[DSSCore]* within the context of the *Section 2.2, “Browser POST”* . Note that these elements were originally defined within the context of signature verification.

The presence of the `<dss:ReturnSignerIdentity>` optional input instructs the DSS server to return a `<dss:SignerIdentity>` optional output as part of the signature creation process.

3.2. Localization

The web application could include the optional input `<dss:Language>` element as defined in *[DSSCore]* section 2.8.3 to indicate the preferred localization to be used by the DSS server as part of the *Section 2.2, “Browser POST”* request.

3.3. SOAP with Attachments

For large documents, the web application and DSS service can use SOAP with attachments *[SwA]* . For larger documents, this can have a very positive impact on the performance of the DSS service. In this case the `<dss:Document>` element looks as follows:

```
<dss:Document ID="doc1">
    <dss:AttachmentReference MimeType="..." AttRefURI="cid:...">
        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>...</ds:DigestValue>
    </dss:AttachmentReference>
</dss:Document>
```

The digest value should always be present given the fact that we sign the SOAP message body.

The DSS server should only use SOAP attachments for the document downloading (see *Section 2.3, “Downloading the signed document”*) when the web application itself used SOAP attachments for the document uploading (see *Section 2.1, “Document uploading”*).

4. Signature Verification

Compared to the eID DSS 1.0.x product, we also made several improvements to the signature verification web service. Here we also allow the usage of SOAP attachments to improve the performance.

The request for a signature verification looks as follows:

```
<soap:Envelope>
    <soap:Body>
        <dss:VerifyRequest Profile="urn:be:e-contract:dssp:1.0">
            <dss:OptionalInputs>
                <vr:ReturnVerificationReport>
                    <vr:IncludeVerifier>false</vr:IncludeVerifier>
                    <vr:IncludeCertificateValues>
                        true
                    </vr:IncludeCertificateValues>
                </vr:ReturnVerificationReport>
            </dss:OptionalInputs>
            <dss:InputDocuments>
                <dss:Document ID="document-id">
                    <dss:Base64Data MimeType="text/plain">
                        ...
                    </dss:Base64Data>
                </dss:Document>
            </dss:InputDocuments>
        </dss:VerifyRequest>
    </soap:Body>
</soap:Envelope>
```

We use the OASIS Profile for Comprehensive Multi-Signature Verification Reports Version 1.0 *[DSSVer]* .

The corresponding SOAP response looks as follows:

```
<soap:Envelope>
    <soap:Body>
        <dss:Response RequestID="the original request id">
            <dss:Result>
                <dss:ResultMajor>
                    urn:oasis:names:tc:dss:1.0:resultmajor:Success
                </dss:ResultMajor>
            </dss:Result>
            <dss:OptionalOutputs>
                <vr:VerificationReport>
                    <vr:IndividualReport>
                        <vr:SignedObjectIdentifier>
                            <vr:SignedProperties>
                                <vr:SignedSignatureProperties>
                                    <xades:SigningTime>
                                        2010-09-13T15:35:49.767+02:00
                                    </xades:SigningTime>
                                </vr:SignedSignatureProperties>
                            </vr:SignedProperties>
                        </vr:SignedObjectIdentifier>
                        <dss:Result>
                            <dss:ResultMajor>
                                urn:oasis:names:tc:dss:1.0:resultmajor:Success
                            </dss:ResultMajor>
                        </dss:Result>
                        <vr:Details>
                            <vr:IndividualCertificateReport>
                                <vr:CertificateIdentifier>
                                    <xmldsig:X509IssuerName>
                                       SERIALNUMBER=200612, CN=Citizen CA, C=BE
                                    </xmldsig:X509IssuerName>
                                    <xmldsig:X509SerialNumber>
                                        21267647932559078400084294942057726232
                                    </xmldsig:X509SerialNumber>
                                </vr:CertificateIdentifier>
                                <vr:Subject>
                                    CN=..., C=BE
                                </vr:Subject>
                                <vr:ChainingOK>
                                    <vr:ResultMajor>
                                        urn:oasis:names:tc:dss:1.0:detail:valid
                                    </vr:ResultMajor>
                                </vr:ChainingOK>
                                <vr:ValidityPeriodOK>
                                    <vr:ResultMajor>
                                        urn:oasis:names:tc:dss:1.0:detail:valid
                                    </vr:ResultMajor>
                                </vr:ValidityPeriodOK>
                                <vr:ExtensionsOK>
                                    <vr:ResultMajor>
                                        urn:oasis:names:tc:dss:1.0:detail:valid
                                    </vr:ResultMajor>
                                </vr:ExtensionsOK>
                                <vr:CertificateValue>
                                    base64 encoded signer certificate
                                </vr:CertificateValue>
                                <vr:SignatureOK>
                                    <vr:SigMathOK>
                                        <vr:ResultMajor>
                                        urn:oasis:names:tc:dss:1.0:detail:valid
                                        </vr:ResultMajor>
                                    </vr:SigMathOK>
                                </vr:SignatureOK>
                                <vr:CertificateStatus>
                                    <vr:CertStatusOK>
                                        <vr:ResultMajor>
                                        urn:oasis:names:tc:dss:1.0:detail:valid
                                        </vr:ResultMajor>
                                    </vr:CertStatusOK>
                                </vr:CertificateStatus>
                            </vr:IndividualCertificateReport>
                        </vr:Details>
                    </vr:IndividualReport>
                </vr:VerificationReport>
                <dssp:TimeStampRenewal Before="2013-11-08T08:49:51.040Z"/>
            </dss:OptionalOutputs>
        </dss:Response>
    </soap:Body>
</soap:Envelope>
```

The signature is uniquely identified by the `<xades:SigningTime>` element. The certificate of the signatory is delivered via the `<vr:CertificateValue>` element.

Via the `<dssp:TimeStampRenewal>` element the web application gets informed by the DSS when the document signatures should be upgraded for long-term validity. See also *Section 5, “Long-term validity of signatures”* .

4.1. Errors

In case the DSS received an unsupported document format, the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

`urn:be:e-contract:dssp:1.0:resultminor:UnsupportedMimeType`

In case the DSS detected an error in one of the signatures , the DSS service returns in `<dss:SignResponse>` a `<dss:ResultMajor>` of `RequesterError` and a `<dss:ResultMinor>` of

`urn:oasis:names:tc:dss:1.0:resultminor:invalid:IncorrectSignature`

5. Long-term validity of signatures

To discuss long-term validity of signatures, we first introduce the following notation:

Self-claimed signing time:

T_s

Time stamp time:

T_{tsa}

Now:

T_{now}

X509 certificate chain of signer:

$X509_s$

X509 certificate chain of time stamp authority:

$X509_{tsa}$

Set of signer certificate status information (can be a combination of CRL and OCSP), collected at time `t` :

$CRL_s(t)$

Set of TSA certificate status information (can be a combination of CRL and OCSP), collected at time `t` :

$CRL_{tsa}(t)$

PKI validation at time `t` (can be in the past) of certificate `X509` with revocation data `CRL` :

`validate(X509, CRL, t)`

Note that $validate(X509, CRL(t_1), t_2)$ will fail if t_2 is too far from t_1 , causing the revocation data $CRL(t_1)$ captured at time t_1 to be already expired at validation time t_2 .

For the signing certificate $X509_s$ it is clear that we want:

$validate(X509_s, CRL_s(T_s), T_s)$

If the signature contains a time stamp, we want

$T_{tsa} - T_s < max\_dt$

with `max_dt` some DSS application specific setting.

For the validation of the time stamp certificate chain $X509_{tsa}$ however, we cannot use:

$validate(X509_{tsa}, CRL_{tsa}(T_{tsa}), T_{tsa})$

Imagine that the TSA gets hacked after time `t` with $t > T_{tsa}$. Then T_{tsa} cannot be trusted anymore. This means that for the validation of the TSA certificate chain we need to use:

$validate(X509_{tsa}, CRL_{tsa}(T_{now}), T_{now})$

Even when using $CRL_{tsa}(T_{now})$ it makes sense to store $CRL_{tsa}(T_{tsa})$ as part of the signature in case the CA of the TSA becomes unavailable due to a major security event, or if you receive a signature that skipped the time stamp renewal and thus for which $CRL_{tsa}(T_{now})$ is also not available anymore. This can happen in practice as CAs are not required to keep expired certificates on their CRLs. Another reason to store $CRL_{tsa}(T_{tsa})$ is because certificates can get suspended for a short period of time. To be able to check whether the $X509_{tsa}$ was suspended during creation of the time stamp at time T_{tsa}, you need to have $CRL_{tsa}(T_{tsa})$ available.

Right before the certificate of the TSA (or one of the intermediate certificates of the corresponding certificate chain) expires, you should capture CRL_{tsa} within the signature. Ideally, after TSA expiration, you should get the guarantee that the TSA private key is destroyed. However such guarantee cannot hold in reality. So right before expiration of the TSA certificate, we must do something else to secure this event.

Let us introduce a second time stamp authority. Hence we have an inner TSA used to create an inner time stamp:

$X509_{itsa}$

and an outer TSA used to create an outer time stamp:

$X509_{otsa}$

The outer TSA certificate chain should be checked via:

$validate(X509_{otsa}, CRL_{otsa}(T_{now}), T_{now})$

If this yields a valid $X509_{otsa}$, then T_{otsa} can be trusted.

The inner TSA certificate can now be checked via:

$validate(X509_{itsa}, CRL_{itsa}(T_{otsa}), T_{otsa})$

This means that when creating the outer time stamp (at time T_{otsa}), we need to capture the revocation data $CRL_{itsa}(T_{otsa})$ of the inner TSA certificate. Even when we already have $CRL_{itsa}(T_{itsa})$ as part of the signature we still need to capture $CRL_{itsa}(T_{otsa})$ as $CRL_{itsa}(T_{itsa})$ might already contain expired revocation data (certificate status information) at time T_{otsa} and thus:

$validate(X509_{itsa}, CRL_{itsa}(T_{itsa}), T_{otsa})$

would otherwise fail.

Again note that keeping track of $CLR_{itsa}(T_{itsa})$ still makes sense to be able to check whether $X509_{itsa}$ was not suspended during the creation of the inner time stamp at time T_{itsa}. Thus ideally we keep track of both $CRL_{itsa}(T_{itsa})$ and $CRL_{itsa}(T_{otsa})$.

This process of time stamp renewal reoccurs every time the outer TSA certificate chain expires.

This process also occurs when one of the used digest algorithms has been shown to be weak for verification purposes. In this case the outer time stamp should use a stronger digest algorithm.

The end result of a signature verification is:

$(X509_s, T_s)$

The above reasoning is very much in favour of the time stamp and document notarization business. Let's approach it from another point of view. Suppose we follow the above time stamp renewal strategy. What does it eventually bring us?

It is clear that the inner most time stamp, called the signature time stamp, is only there to acknowledge the self-claimed signing time via:

$T_{tsa} - T_s < max\_dt$

Suppose something happens with this time stamping authority TSA. The only meaningful reaction to such an event would be to redo the time stamp. However, we probably won't be able to maintain:

$T_{tsa} - T_s < max\_dt$

And hence cannot assert the self-claimed signing time T_s anymore. So what's to point in redoing this time stamp anyway?

Similar for the outer most time stamp authority `otsa` , we have that redoing an outer time stamp in case of a security event with its TSA, will make the following security check impossible:

$validate(X509_{itsa}, CRL_{itsa}(T_{otsa}), T_{otsa})$

Since we just witnessed a security event with the inner TSA `itsa` at time `t` with $t > T_{otsa}$. So again, there is little reason in redoing this outer most time stamp.

The irony of it all is that the only possible strategy to cope with these events is to renew the time stamps even more frequently, and hence is even more in favour of the TSA and notarization business. Here we would also need to always create at least two time stamps instantly with zero correlation as it comes to security events. Something impossible to guarantee.

It is a fact that long-term validity of signatures is economically and practically unfeasible.

So right before the signature time stamp expires you simply capture

$CRL_{tsa}(T_{now})$

one last time and you end the sequence there. If you're certain that the CA of the TSA will never remove expired certificates from its CRLs, you even do not have to do anything.

Executive summary: if you lose a TSA, you're foobar anyway.

6. XML Schema

The XML schema for the Digital Signature Service protocol is given below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="urn:be:e-contract:dssp:1.0"
    elementFormDefault="qualified" attributeFormDefault="unqualified"
    xmlns:tns="urn:be:e-contract:dssp:1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <element name="TimeStampRenewal" type="tns:DeadlineType" />

    <complexType name="DeadlineType">
        <attribute name="Before" type="xs:dateTime" use="required" />
    </complexType>

</schema>
```

A. Digital Signature Service Protocol Specifications License

B. Digital Signature Service Protocol Project License