Technical Specification for LSS

Table of contents

Version history

4th March 2014	Version 0.93	TSS
31th January 2014	Version 0.92	MSP
10th January 2014	Version 0.91	MSP
20th December 2013	Version 0.6	MSP
16th December 2013	Version 0.5	MSP

1 The purpose and audience of the document

The purpose of this document is to provide the technical specification for the implementation of NemID Erhverv TU JavaScript functionality towards customers with central signature servers (LSS), henceforth referred to as LSS for NemID.

The document is aimed at the LSS suppliers who are to implement the functionality into their LSS solution

Summary of all documents in the LSS for NemID Package:

Implementation documentation

- Technical specification for LSS
- Implementation guide for LSS

Test documentation

- Guidelines on the use of LSS for NemID test tools
- Testprocedures for LSS

Solution documentation

- Requirements feedback form for LSS
- End-customer documentation for LSS

2 Introduction

The purpose of the NemID service provider package for Central Signature servers (LSS for NemID) is to provide a JavaScript based integration between service providers (SP) and employees at organizations, who have their NemID for business stored **on a central signature server hosted on their enterprise LAN**.

As a supplier of LSS products it is possible to integrate to the LSS for NemID. This makes it possible for employees at companies with the LSS to use NemID for business from JavaScript enabled devices such as tablets, smartphones and ordinary computers. Note that service providers may and may not choose to support the LSS for NemID functionality in their services.

This document states the requirements for LSS suppliers to integrate test and document their integration to LSS for NemID.

For a general introduction to NemID and NemID for business consult the current service provider package (TU-pakke) from DanID[1]. For the rest of this document, knowledge of the general concept of NemID and NemID for business, as found in the current service provider package, is expected.

[1] https://www.nets-danid.dk/tu-pakke

3 Solution architecture

The purpose of the LSS for NemID is to provide the ability for employees of organizations with LSS to authenticate towards service providers and to digitally sign documents in formats Text, HTML, XML and PDF.

To support this functionality, the LSS supplier needs to implement authentication and signing capabilities towards their own LSS backend.

The overall architecture is illustrated below.

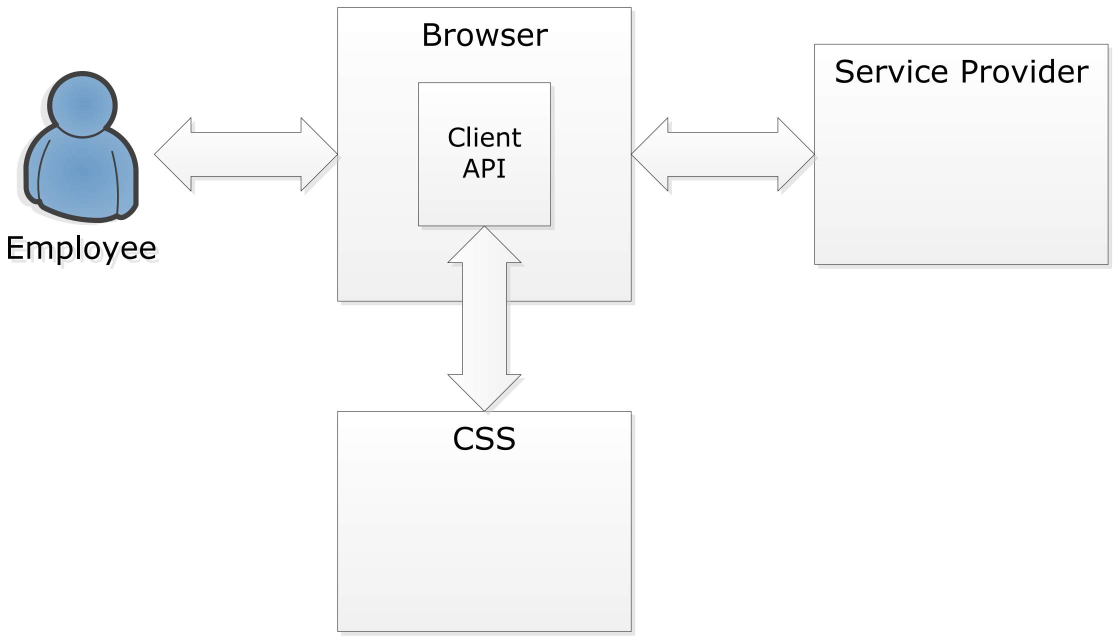


Fig 1: Overall architecture

3.1 IFrame integration

The JavaScript LSS for NemID client is integrated with the service provider's page using an <iFrame> element, which enables a web page to allocate a segment of its area to another page. This is a change from the Java applet client, where a Java applet was loaded as a page element and works similar to NemID JavaScript.

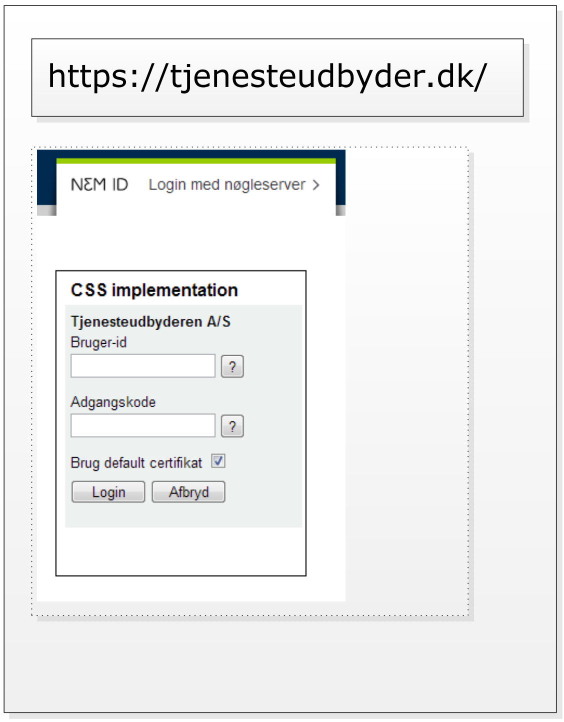


Fig 2: The iFrame

An <iFrame> element does not allow its content to expand beyond its borders, which necessitates that an area sufficient for every possible screen size is allocated when it is created.

To provide consistency with the other NemID solutions the iFrame has a minimum width and height matching the requirements of the "Limited Mode" setting of NemID JavaScript both for authentication and signing.

In shorter terms, this means that the iFrame must be created with a width of at least 200 pixels and a height of at least 250 pixels, as illustrated in Figure 1.

As a LSS supplier, you must expect that the iFrame setup by the service provider conform to the minimum requirements.

For signing flows, it is highly recommended, that you utilize the entire allocated space inside the iFrame in a dynamic way, such that the service provider is able to scale the frame according to the screen-size of the users device.

[NOTE: It might be difficult to create a signing flow visible-acceptable at 200*250 pixels. We a-wait how the NemID JavaScript recommendations form during the next couple months]

4 Flow (logon and signing)

The general flow is illustrated below

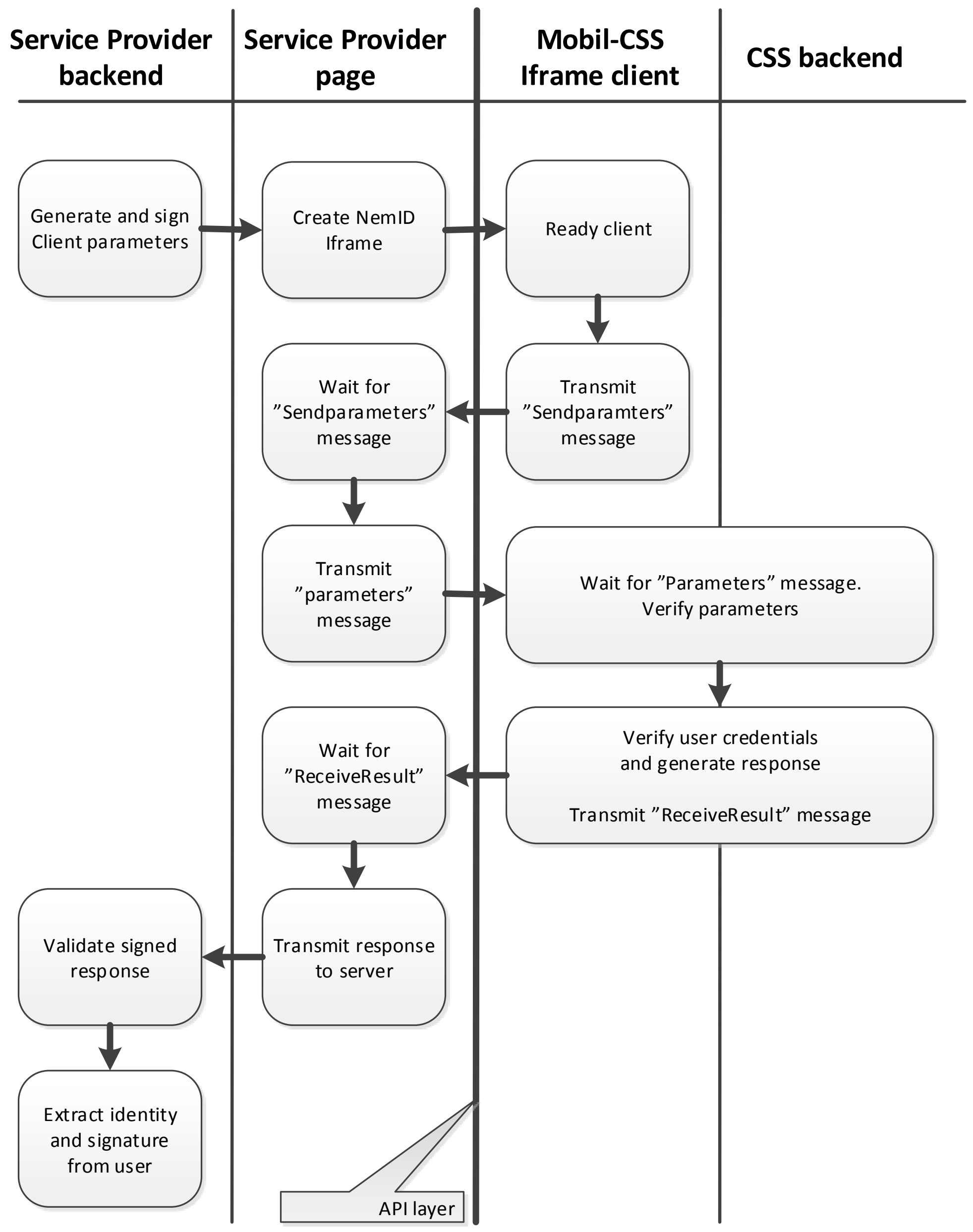

Service Provider backend
Service Provider page
Mobil-CSS Iframe client
CSS backend
Generate and sign Client parameters
Create NemID Iframe
Ready client
Wait for "Sendparameters" message
Transmit "Sendparamters" message
Transmit "parameters" message
Wait for "Parameters" message. Verify parameters
Wait for "ReceiveResult" message
Verify user credentials and generate response
Transmit "ReceiveResult" message
Validate signed response
Transmit response to server
Extract identity and signature from user
API layer

Fig 3: General flow

5 Bootstrapping

To enable clients and service providers to utilize local LSS-installations, the solution is bootstrapped by the service provider by starting up an iFrame pointing to a fixed URL. The DNS on the customer's internal network must be configured to make the bootstrap address point to the local LSS installation.

Address used by the service providers to bootstrap the iFrame:

Bootstrap Address	https://lss-for-nemid-server.dk/<random digits>

Note that the <random digits> postfix of the URL must be a constantly changing number such as system time in milliseconds. Its purpose is to prevent caching of resources in the client and can be ignored by the LSS supplier. LSS suppliers must handle this form of the request, but may assume that <random digits> are a series of numbers (0-9) without a trailing slash.

6 Secure Sockets Layer (SSL)

To ensure authenticity for the clients towards the possible LSS servers they might be contacting, the iFrame will be run over SSL/https.

To enable this solution, client browsers need to trust the certificate used for the https-connection.

To enable trust on the devices used with this solution, the LSS-organization administrating the devices must establish a custom trust Certificate Authority (CA), or have their LSS supplier help them on this matter. This CA will be used for issuing SSL-certificates for the Bootstrap Address.

6.1 Defending against malicious LSS suppliers

By setting up the iFrame using SSL, content will only be shown to users inside the iFrame, if the browser trusts the certificate used. This ensures that only trusted LSS suppliers are able to present content inside the iFrame.

If for instance a user is connected to his work-network through a VPN-tunnel from a public wireless network, say an airport, and the VPN connection terminates without the user noticing. Then, attempting to use LSS for NemID on some service provider's page, the user browser resolves the **Bootstrap Address** for the iFrame on the untrusted network, which on that network could be spoofed by any malicious party.

Without SSL/https, the user has no way of knowing whether the content inside the iFrame comes from his or hers local LSS supplier or whether it is from a malicious party.

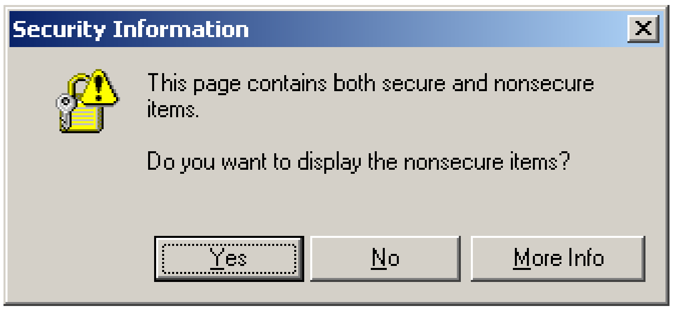
With SSL/https, the malicious party will not be able to setup a trusted connection and the content inside the iFrame will be blocked by the user's browser.

6.2 Encryption of user credentials

Using SSL/https add encryption to the information sent between the LSS supplier and the user, and thus protects the information passed from the user to the LSS supplier, in particular the credentials passed from the user to the LSS supplier.

6.3 Non-secure content warning messages

Hosting an iFrame not running https inside a page that is running https will provoke a warning message in all major browsers.


Security Information

This page contains both secure and nonsecure items.

Do you want to display the nonsecure items?

This is avoided by running the iFrame under SSL/https.

7 JavaScript API

The communication between the service provider and the LSS supplier across the boundaries of the iFrame and the service provider's page is done using the HTML5 Web Messaging API.

The flow starts, when the service provider loads an iFrame with the **Bootstrap Address** on a page and hooks up a JavaScript function as an event listener. When the LSS supplier is ready, a "SendParameters" command is passed to the service provider page. The LSS supplier then awaits the "Parameters" command containing the parameters specifying the flow and content hereof.

When the LSS supplier is done processing user input and creating an XML-DSig or error code, the result is passed to the service provider page using the “ReceiveResult” command.

To illustrate:

1) The iFrame is started. Service provider awaits “SendParameters” command.
2) LSS supplier passes the “SendParameters” command to the service provider page.
3) The service provider sends the “Parameters” command containing all required parameters.
4) The LSS supplier sends the “ReceiveResult” command containing the result of the operation.

All calls using this API are done by sending a message containing the command name, ex. “SendParameters” and a content value defined by a JSON object mapping containing parameter name/value-pairs.

8 Start-up and handling responses

The following section contains an overview of the communication flow between a service provider and the LSS supplier, here illustrated by a boilerplate for the iFrame entry-point at the LSS supplier.

The nature of the HTML5 Web Messaging API makes it necessary to use a synchronization message to signal to the service provider page when the client is ready. This is what is referred to as the SendParameters message above. The service provider must wait until it receives that message, and then transmit the parameters to the LSS supplier.

The following JavaScript runs on the LSS supplier’s page, and is an example of how to handle the message exchange.

```
 <script type="text/javascript">
       function onmessage(e) {
           var event = e || event;

           var message = JSON.parse(event.data);
           if (message.command === "Parameters") {
              //This command contains all parameters from the
              //service provider
           }
       }

       if (window.addEventListener) {
           window.addEventListener("message", onmessage);
       } else if (window.attachEvent) {
           window.attachEvent("onmessage", onmessage);
       }

       var postMessage = {};
       postMessage.command = "SendParameters";
       postMessage.content = "{
                                 "API_Version":"1.0.0.0",
                                 "LSS_ID":"LSS-DEMO",
                                 "LSS_Version":"1.0.0.1"
                               }";
       parent.postMessage(JSON.stringify(postMessage), "*");

 </script>


/* When ready to send back result or error */

<script type="text/javascript">
           var postMessage = {};
            postMessage.command = "ReceiveResult";
            postMessage.content = "{
                                        "Status":"CSS000",
                                         "Signature":"[XML in Base64]"
                                      }";
            parent.postMessage(JSON.stringify(postMessage), "*");
</script>
```

The code attaches an event listener to the message event, which is fired whenever the document receives a message from other documents [Web Messaging]. The event listener, in this case the function named **onmessage**, handles the following cases:

- On load of the LSS supplier’s entry page inside the iFrame, the “SendParameters” command is fired to signal the service provider to send the parameters for the request.
- A handler for the “Parameters” command which receives the signed parameters as JSON Object.

- When done processing the parameters the LSS supplier interacts with the user and sends the "ReceiveResult" command to the service provider with an error or the XML-DSig-signature.

8.1 IFrame presentation to user

The service provider awaits the "SendParameters" command and parameters and does not present a visible iFrame to the user before the parameters has been parsed and verified. The user will not be able to interact with the LSS supplier before the service provider has received and accepted the "SendParameters" command.

9 Parameters

The JavaScript messages sent between the LSS supplier and service provider is passed as a JSON object. Each parameter consists of a name and a value, both of which are strings. An example of parameters in a JSON object is given below.

```
{
 "SP_CERT":"….P6zBVn6bnYfDSzsZNAhb",
 "CLIENTFLOW":"login",
 "TIMESTAMP":"MjAxMy0xMi0wNiAxMzo1NDo0MSswMTowMA==",
 "REQUESTISSUER":"U2lnbmF0dXJncnVwcGVuIA==",
 "LANGUAGE":"DA",
 "PARAMS_DIGEST":"OcjHvzdfK/kch..tdW1lqGycbQpMMp94/Y=",
 "DIGEST_SIGNATURE":"X6oUkRZu+…cUjGhexQ6uqPtAUwpqGHIifpj3in4nw="
}
```

The ordering of the parameters in the JSON object holds no significance. All parameter names are case-insensitive. Some values, e.g. base64 encoded strings or URLs, are case-sensitive.

The set of parameters and their respective allowed name-value-pairs form the API. Current version of the API is reflected in the API_Version parameter in this section.

9.1 Parameters to send in the "SendParameters" call

Name	Description	Allowed values
API_VERSION *Mandatory*	Version of implemented API. Service providers may use this to determine the version of the LSS supplier's	Supporting this version of the API: 1.0.0.0

	implementation.	
LSS_ID	LSS specific identifier	Id chosen by the LSS supplier
LSS_VERSION	LSS specific version	x.y.z.v
PDF _SUPPORTED	LSS support for PDF signing. **Default: TRUE**	**TRUE or FALSE**

9.2 Parameters received in the "Parameters" call

The following tables contain a description of the parameters.

Name	Description	Allowed values
CLIENTFLOW *Mandatory*	Determines which flow to start	• LOGIN • SIGN
LANGUAGE	Client language	• DA Danish - default • EN English
TIMESTAMP *Mandatory*	Current time when generating parameters. Recommended, that LSS suppliers reject messages older than 3 minutes. Example: 2013-12-17 13:33:47+0100	Current time in UTC expressed as one of: yyyy-MM-dd HH:mm:ssZ (Java) yyyy-MM-dd HH:mm:sszzz (.Net) Epoch Milliseconds since 1970-01-01 00:00:00
REQUESTISSUER *Mandatory*	The "logonto" parameter from OpenSign. This parameter will be part of the resulting XML-DSig. The LSS supplier must show this value inside the iFrame.	String. Typical name of service provider.
SIGN_PROPERTIES	Base64 encoded XML	Refer to the general service

	formatted XML-DSig properties to be included in signed response	provider documentation on how to format signing properties. Only applicable in a signing context.
SIGNTEXT	Base64 encoded representation of the actual text signed by the user. Can be in formatted as specified by SIGNTEXT_FORMAT	Only applicable in a signing context.
SIGNTEXT_FORMAT *Mandatory for all signing flows*	The format of input given by SIGNTEXT parameter	• Text • HTML • XML • PDF Only applicable in a signing context.
SIGNTEXT_MONOSPACEFONT	Indicates that plaintext should be rendered with a mono-spaced font (to allow for indention based formatting)	• TRUE Mono space font is used for plaintext Any other value Default font is used. Only applicable in a signing context.
SIGNTEXT_REMOTE_HASH *Mandatory if SIGNTEXT_URI is specified*	Base64 encoded SHA256 hash of the remote PDF document	Only applicable in a signing context.
SIGNTEXT_TRANSFORMATION *Mandatory if SIGNTEXT_FORMAT is XML*	Base64 encoded XSLT style sheet used to transform the XML sign text into a HTML document	Only applicable in a signing context if SIGNTEXT_FORMAT is XML.
SIGNTEXT_TRANSFORMATION_ID	Identifier for XML stylesheet	Only applicable in a signing context.
SIGNTEXT_URI	URI used to load a PDF document asynchronously	URI must be well formed. Only http or https are supported
SP_CERT *Mandatory*	Base64 encoded DER representation of the certificate used identifying the OCES service provider	Certificate must be issued by a Nets-DanID trusted Certificate Authority
PARAMS_DIGEST *Mandatory*	Base64 encoded SHA256 Digest of normalized parameters	See section 10 for detailed documentation

DIGEST_SIGNATURE *Mandatory*	Base64 encoded Signature of normalized parameters	See section 10 for detailed documentation

9.3 Parameters to send in the "ReceiveResult" call

Name	Description	Allowed values
Status *Mandatory*	Status code for the service provider.	See section 13 for possible error codes.
LSS_SUPPORTMESSAGE *Mandatory*	LSS specific text describing contact information for the local IT-Helpdesk or similar service at the LSS provider. This message can be used by the service provider to provide the user with a useful message on various errors. En example could be when the user's certificate is revoked. "Contact your local helpdesk at xxx…"	The service provider assumes that the text is clear-text without any special formatting. **The value must be Base64 encoded.** The service provider may assume that this value is less than 250 characters and may choose to cut of text longer than this when displaying it to the user.
Signature *Mandatory if Status = LSS000*	Base64 encoded XML-DSig	Valid XML-DSig containing the signed document or signed login request
Status_Text *Mandatory if Status = LSSERR001 OR Status = LSSERR002*	Custom status text, which may be used for logging and error handling at the service provider	The service provider assumes that the text is clear-text without any special formatting. **The value must be Base64 encoded** For LSSERR002 the service provider may assume that this value is less than 250 characters and may choose to cut of text longer than this when displaying it to

		the user.

10 Validating parameters

The LSS supplier must run through all received parameters, and should return an error, if an unexpected parameter name or value is encountered.

The service provider is required to supply a valid signature on the parameters along with their certificate, as described in the next section. Validation of the signature is an optional step available to the LSS supplier, but a valid signature is the only way a LSS supplier is able to determine the identity of the service provider. The certificate supplied bears information about the service provider such as company name and CVR-number.

The required **Timestamp** parameter can be used to ensure, that requests are not too old. A three minute window is the recommended approach.

Validating the signature and the **Timestamp** enables the LSS supplier to log a "proof" that the given service provider requested the specific service at the given time.

10.1 Parameter integrity

To ensure the integrity of the parameters in transit between the service provider and the NemID JavaScript client, they must be signed by the service provider.

The process for securing the parameters is

1. The service provider collects the list of parameters. The list is normalized (see section 10.2) into a string, and the SHA-256 digest value of the string's UTF-8 representation is calculated.

2. The digest value is signed by the service provider. The signature is performed using the VOCES certificate, which is associated with his service agreement with Nets DanID. The signature algorithm to be used is RSA SHA-256.

3. The Base64-encoded value of the digest and the signature are added as the parameters PARAMS_DIGEST and DIGEST_SIGNATURE.

4. All parameters are collected in a JSON-message and sent to the LSS supplier.

5. The LSS supplier reads the parameters and normalizes them, excluding the digest value and the signature parameters. The digest value is verified by comparing the calculated digest with the supplied.

6. The LSS supplier verifies the signature using the certificate of the service provider supplied in the SP_CERT parameter. The certificate must be a VOCES or FOCES issued by DanID.

10.2 Parameter normalization

The digest of the client parameters are calculated from a normalized version of the parameters.

The process for normalizing the parameters is

1. The parameters are sorted alphabetically by name. The sorting is case-insensitive.

2. Each parameter is concatenated to the result string as an alternating sequence of name and value: name1 || value1 || name2 || value2 || … || name**n** || value**n**

11 Authentication

An authentication flow is initiated by setting the CLIENTFLOW parameter to login.

12 Signing

The support for signing is the same as supported by the other NemID solutions. Consult the current service provider package (TU-pakke) from Nets DanID[2] for general documentation on the supported signing types and validation.

It is important that the signing-text included in the XML-DSig returned to the service provider contains the exact value received in the

[2] https://www.nets-danid.dk/tu-pakke

“signtext” parameter. Furthermore, validation and presentation of the text/document to be signed, is the responsibility of the LSS supplier.

If a signing operation is unsuccessful, an error code is returned to the service provider. The error code is given to the service provider base64 encoded. Sections 13.4 and 13.5 contain a list of the error codes a service provider may receive from a signing operation.

Validation of the signed result are performed with the general DanID OOAPI tools and described in the current service provider package.

12.1 Rendering of signature flows

It is the responsibility of the LSS backend to validate that the sign texts received for signing flows conform to the specifications given in DanID’s service provider documentation for the specific signing flows (i.e. plain text, HTML, XML and PDF).

Rendering of the document to be signed is likewise the responsibility of the LSS backend and not the service provider.

It is expected that the PDF signing flow should be implemented using the open source JavaScript toolkit PDF.js. Currently it is not mandatory to support this flow.

13 Response

The response to the service provider is sent in the Web Messaging call "ReceiveResult". The status code is mandatory.

13.1 XML-DSig result

The service provider expects a signed XML-DSig when the flow is successfully completed, both for login and signing flows.

The xml should conform to the XML-DSig standard, and thus validate in a generic XML-DSig validator. Furthermore, the signature has added requirements to conform to the structure of other NemID-generated signatures. As a minimum the latest version of OOAPI should validate the signed xml.

13.2 Response codes

An error code is returned to the service provider, if a client operation fails to complete successfully. The error code should be used to assist the user in how to remedy the situation and accomplish what he set out to do.

The list contains 4 categories: General error codes, error codes related to authentication, error codes relating to signing and LSS specific error codes. If possible it is advised to use existing error codes from the NemID/"Digital Signatur" platform, as service providers most likely already have error handling for those in place.

13.3 General error codes

These error codes are general to the client functionality and can be received regardless of which operation the client was supposed to do.

Error code	Cause of error
APP001	The client calculated the digest of its parameters, and it did not match the digest that was submitted in the PARAMS_DIGEST parameter.
APP007	Returned by the client if a mandatory

	parameter is missing, or if an unrecognized parameter has been received.
APP008	Returned by the client if an invalid combination of parameters has been received.
CAN002	The user chose to cancel the operation by pressing the cancel button. This error is not transmitted if the user navigates away from the page containing the client, e.g. by closing the browser window or clicking a link.
SRV006	The server lost the session it had established with the client. This may occur if the user leaves the client open for a prolonged stretch of time without interaction.
SRV003	The time stamp of the authentication request was not within the allowed time span.

13.4 Signing error codes

Error code	**Cause of error**
APP002	The sign text was illegal, e.g. the HTML document contained illegal tags or the PDF document did not match its hash.

13.5 LSS specific error codes

The following codes may be returned during LSS operations.

Error code	**Cause of error**
LSS000	This code is used to signal success.
LSSERR001	When this code is returned, a custom error-text has been supplied in the STATUS_TEXT parameter. The text should not be presented to the user.
LSSERR002	When this code is returned, a custom error text has been supplied in the STATUS_TEXT. The text should be presented to the user. The service provider assumes that the text is clear-text without any special formatting.

	The service provider may assume that this value is less than 250 characters and may choose to cut of text longer than this when displaying it to the user.
LSSPDF001	The LSS backend does not support PDF signing.
LSSAUTH001	Number of allowed authentication attempts exceeded.
LSSAUTH002	The user is not able to authenticate.
LSSLCK001	The user entered incorrect credentials too many times is now locked temporarily.
LSSLCK002	The user entered incorrect credentials too many times is now locked permanently. The user must contact the administrators for the LSS-solution to resolve the problem.
LSSSRV001	The signature on the client parameters could not be verified.
LSSGLB001	This is normally not returned by any LSS supplier. This is reserved for the case, where a global endpoint for the Bootstrap Address has been setup and the user resolves to this. This typically means that the user is not connected to a local network with a local DNS-entry for the Bootstrap Address, and thus gives the service provider a way to fail early.
LSSJSN001	Error parsing JSON object

Appendix A: References

[XMLDSIG]	XML Signature Syntax and Processing (Second Edition) http://www.w3.org/TR/xmldsig-core/
[XMLENC]	XML Encryption Syntax and Processing http://www.w3.org/TR/xmlenc-core/
[RFC 4051]	Additional XML Security Uniform Resource Identifiers (URIs) http://www.ietf.org/rfc/rfc4051.txt
[PKCS1]	PKCS #1: RSA Cryptography Specifications 2.0 http://tools.ietf.org/html/rfc2437#page-13
[Web Messaging]	HTML5 Web Messaging http://www.w3.org/TR/webmessaging/
[JSON]	JavaScript Object Notation http://www.ietf.org/rfc/rfc4627.txt http://www.json.org/
[SOP]	Same Origin Policy http://en.wikipedia.org/wiki/Same_origin_policy
[CORS]	Cross-origin Resource Sharing http://en.wikipedia.org/wiki/Cross-origin_resource_sharing
[XDRO]	XDomainRequest object http://msdn.microsoft.com/en-us/library/ie/cc288060(v=vs.85).aspx http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx