General Technical Specification for LSS for NemID

Table of contents

Version history

December 22th, 2016	Version 2.0	TSS
August 31, 2016	Version 1.5	JGB
October 30, 2014	Version 1.4	JGB
September 26, 2014	Version 1.3	TSS
August 12, 2014	Version 1.2	TSS
4th April 2014	Version 1.1	JGB
28th March 2014	Version 1.0	TN
27th March 2014	Version 0.94	TN
23rd March 2014	Version 0.93	TSS
13th March 2014	Version 0.92	BS
5th March 2014	Version 0.91	TSS
2014	Version 0.9	TSS
2014	Version 0.1	MSP

1 Changes in version 2.0

- ORIGIN parameter changed status from MANDATORY to OPTIONAL
- LSS_SUPPORTMESSAGE removed from specification
- Unknown parameters should be silently ignored by LSS backends
- Updated support for browser caching prevention
- Introduced the https://lss-for-nemid-server-test.dk URL in TEST
- LSS errorhandling should be handled inside the LSS iframe by the LSS backend and not by the service provider. Removed the following errorcodes in this process: LSSAUTH002, LSSLCK001, LSSLCK002
- Issuance of certificates added to the specification

2 Introduction

The purpose of the LSS for NemID service provider package is to provide an integration between service providers (SP) and employees in organizations, who have their NemID for business stored on a local signature server (LSS), hosted on their enterprise LAN.

This makes it possible for employees in companies with the LSS, to use NemID for business from JavaScript enabled devices such as tablets, smartphones and ordinary computers.

This document provides the technical specification for the LSS for NemID setup and provides the technical implementation reference for service providers as well as for the technical staff in organizations implementing an LSS for NemID backend.

The reader is assumed to be familiar with the general concepts of NemID and NemID for Business as presented in the current service provider package (TU-pakke) from NETS DanID [TU].

As of fall 2016 the LSS for NemID functionality is included in the service provider package from NETS DanID.

3 Solution architecture

The purpose of the LSS for NemID is to provide the ability for employees in 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 service provider needs to setup and handle communication with the users LSS back-end through an HTML *iframe* using JavaScript.

The overall architecture is **illustrated** below.

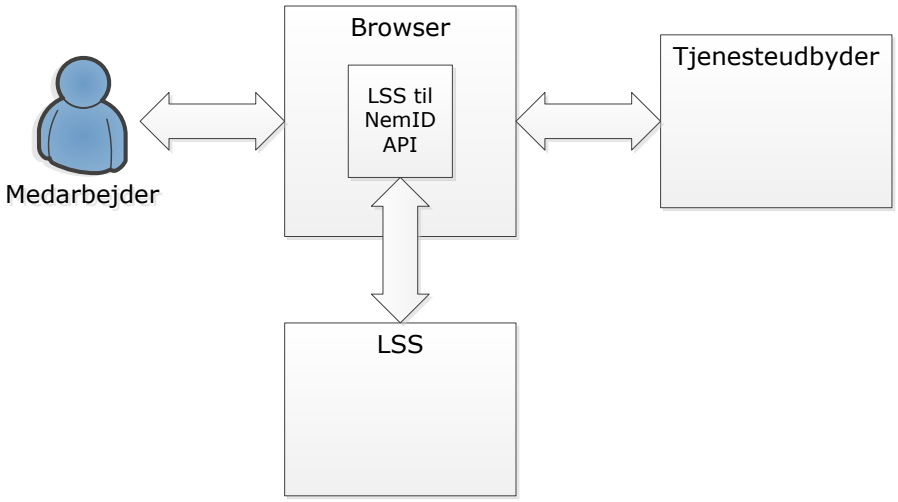


Figure 1: Overall architecture

3.1 LSS Client integration

The JavaScript LSS for NemID client (LSS 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 differs from the Java applet client, where a Java applet is loaded as a page element and works like NemID JavaScript.

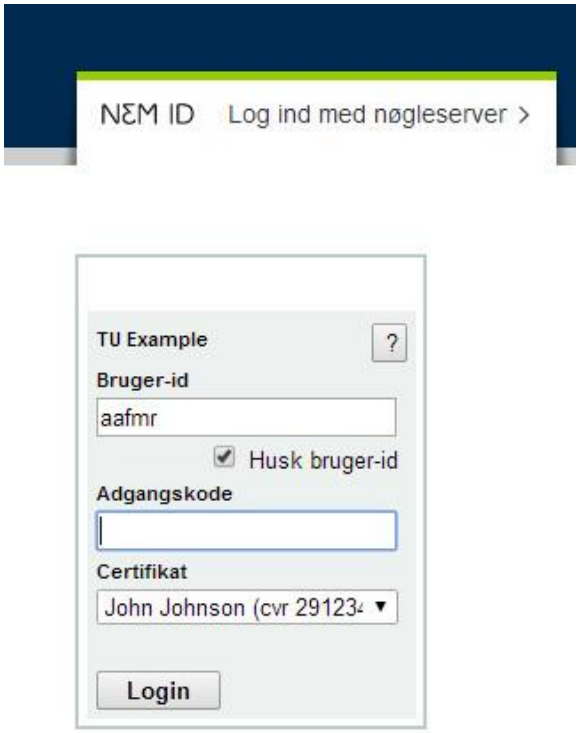


Figure 2: The iframe

An *iframe* element has a fixed size, which requires allocation of sufficient room for every possible screen size, when it is created.

The *iframe* element must be setup with a width of at least 200 pixels and a height of at least 275 pixels, as illustrated in Figure 2. This is the same minimum width as the NemID JS *iframe* in Limited Mode, but has an increased height from 250 to 275 pixels.

For signing flows, it is recommended, that the LSS backend use the entire allocated space inside the *iframe* element in a dynamic way, so that the service provider can scale the frame to the screen size of the user's device.

Screen reader tools depend on the title attribute for describing iframes. An appropriate title could be "LSS for NemID".

3.2 Third-party cookies

When an *iframe* is setup with a source from a domain outside the service provider's domain, any cookies exchanged through the frame are considered third party cookies – and this setup is exactly that of NemID for LSS when using the LSS Client.

The support for third party cookies is not guaranteed across available browsers.

It is required that LSS suppliers implement their backend service without the use of cookies to accommodate all browsers.

The HTML5 Web Storage API could be utilized to store client-side values like the username.

Using *cookie-less* sessions could be utilized to provide a session-based back-end without using cookies.

3.3 LSS Client URL

To enable the LSS Client, the service provider must setup an *iframe* element using the LSS Client URL.

LSS Client URL PROD	**https://lss-for-nemid-server.dk/**
LSS Client URL TEST	**https://lss-for-nemid-server-test.dk/**

To prevent browser caching of the content inside the LSS Client iframe one of the following strings is appended to the LSS Client URL:

- **https://lss-for-nemid-server.dk/<random-digits>**
- **https://lss-for-nemid-server.dk/?t=random**

The purpose of the random-digits or the random postfix of the URL is to supply a random constantly changing number, such as system time in milliseconds to prevent caching of resources in the client and should be ignored by the LSS supplier. LSS suppliers must handle this form of the request, but can assume that **<random-digits>** are a series of digits without a trailing slash.

Example of valid URLs:

https://lss-for-nemid-server.dk/1395749519

https://lss-for-nemid-server-test.dk/?t=22333345223

Example of invalid URL's:

https://lss-for-nemid-server.dk/1395749519/

https://lss-for-nemid-server.dk/aabbccdd

To ensure authenticity of the LSS servers towards clients the LSS Client MUST be established over SSL/https.

For this solution to work, client browsers need to trust the certificate used for the https connection.

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

If the user's browser does not have the appropriate SSL-trust, the *iframe* content is blocked by the browser and consequently the LSS backend will not respond to the initialization.

3.3.1 LSS Client iframe attributes

This section describes LSS Client *iframe* attributes that will have a profound effect on the user experience.

The SP should carefully consider how to set these attributes.

The *allowfullscreen* attribute

This attribute takes values *true* or *false.* When set to true, this attribute will allow the *iframe* to display its contents in full screen mode.

It is recommended, that the value *true* is used for signing flows.

The *scrolling* attribute

This attribute takes values *yes* or *no.* When set to *yes* the user will be able to scroll the contents within the *iframe* and the browser may or may not use space for displaying scrollbars.

For that reason it is recommended that this attribute is set to *no.*

3.3.2 Hybrid app integration

Implementing a native application for a mobile platform (i.e. iOS, Android, Windows etc.) based around a web container and web content is often referred to as a *hybrid app.*

The LSS Client is a web based client only working through the HTML5 JavaScript based Web Messaging API. To integrate with the LSS Client from a native application the application must be able to serve web content, setup the LSS Client *iframe* and so forth and naturally makes any app integrating with the API a *hybrid app*.

Creating a hybrid app that interacts with the LSS Client is done in the same manner as from a regular web application and thus no additional documentation is provided for the *hybrid app* scenario.

The LSS Client *iframe* must still be created and JavaScript enabled to communicate with the LSS back-end through the LSS Client.

The *iframe* minimum size is still 200 * 275 pixels.

4 Logon and signing flow

The general flow between the service provider and the LSS is illustrated below.

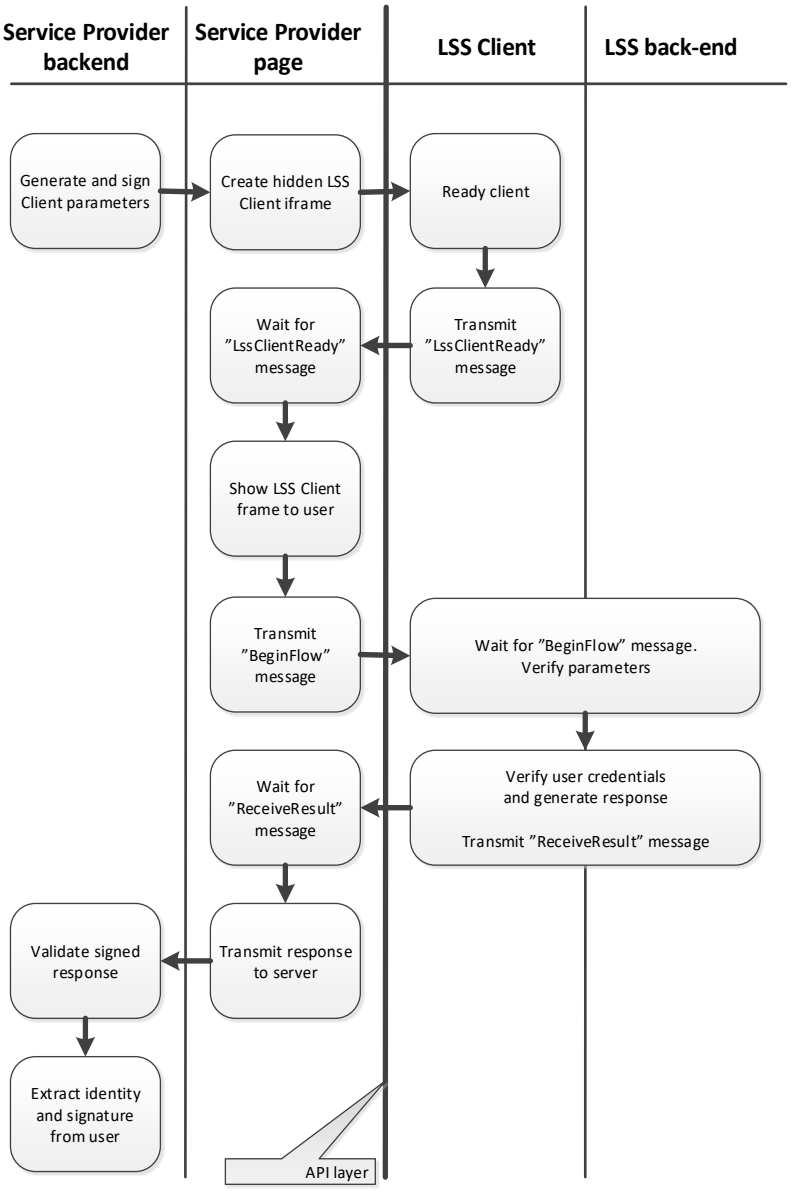


Figure 3: General flow

The flow is initiated when the service provider decides which flow to initiate – signing or logon – and generates the appropriate set of

parameters (content to sign etc.) initializing that flow. Details of all API invocations are given in Section 6.

The service provider then loads the LSS Client at the LSS Client URL.

When loaded properly, the LSS Client will send the *LssClientReady* command to the service provider page.

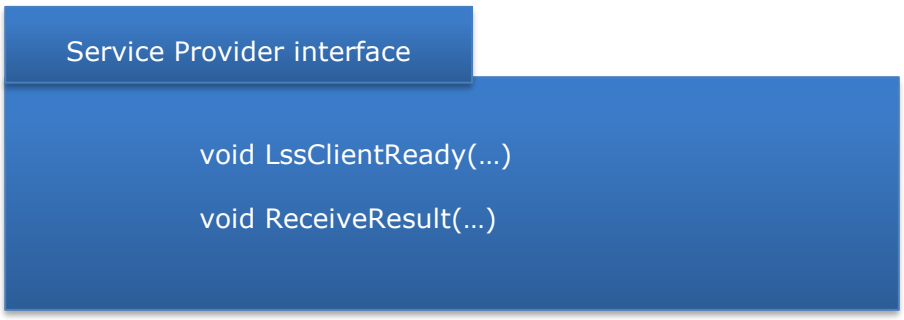
When the service provider receives *LssClientReady* the command verified. If found valid, the LSS Client *iframe* is made visible to the user, i.e. the user is not able to interact with the LSS before the service provider has received and accepted the *LssClientReady* command.

The LSS then awaits the *BeginFlow* command containing the parameters, specifying the desired flow and content hereof.

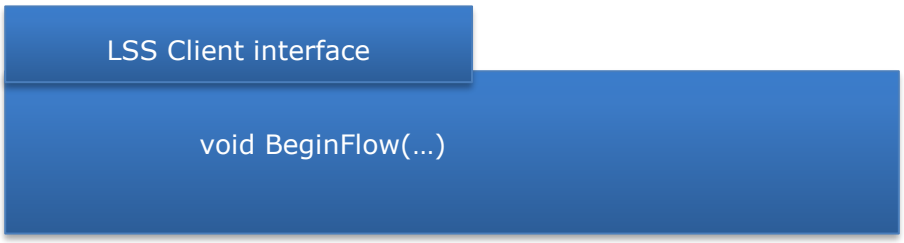
When the LSS is done processing user input and has created either an XML-DSig message or error code, the result is passed on to the service provider page, using the *ReceiveResult* command.

4.1 Communication API

The API defining the communication channel between the service provider and the LSS can thus be expressed by two interfaces, one implemented by the service provider and one by the LSS Client:



a



The following section will outline how these interfaces are implemented.

In Section 6 the method parameters illustrated by dots above are detailed.

4.2 Initialization error handling

When the LSS is unreachable, the normal flow described above will fail when the service provider page attempts to load the LSS Client.

The flow is sketched below:

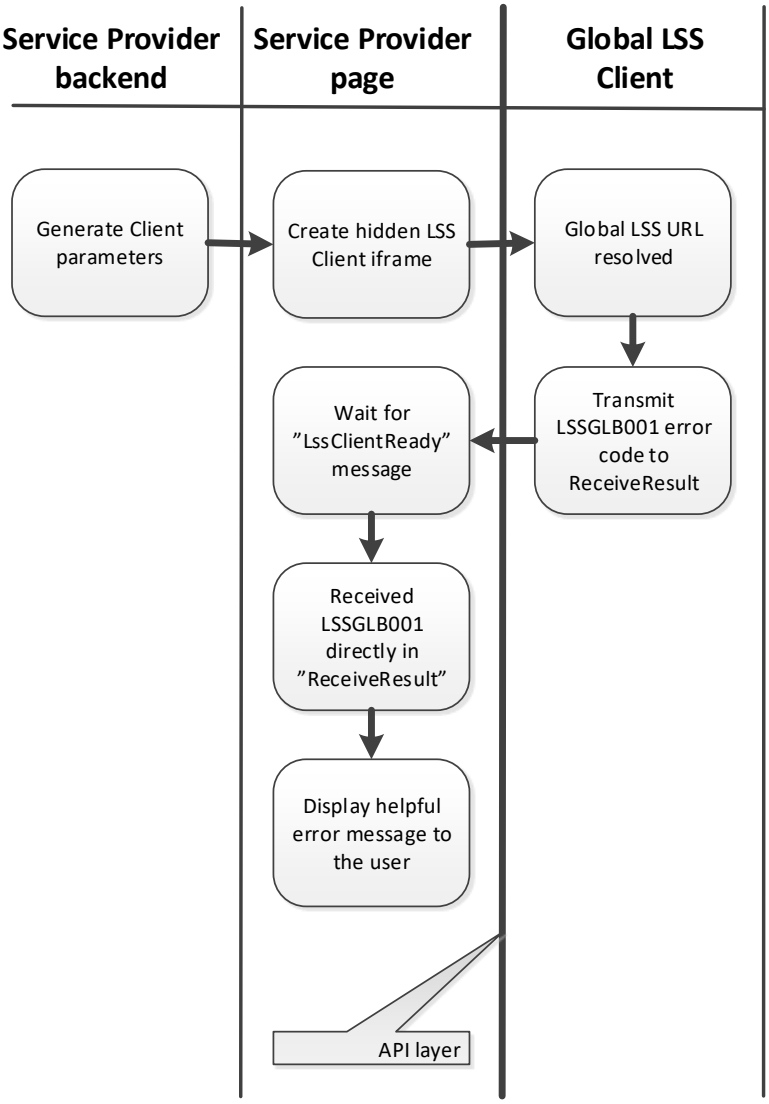


Figure 4: Global flow

Since the user device is not on the company LAN, the DNS lookup for *lss-for-nemid-server.dk* will resolve to the Global LSS Client or fail to resolve entirely.

A Global LSS Client is setup to provide a fast indication to the client, that the user is unable to locate a local LSS service.

The most common causes for a non-responding LSS backend are listed here:

- Browser does not trust the SSL certificate used for the LSS Client URL.

- Neither the global LSS address nor the LSS is available to the user.
- Local LSS backend is irresponsive. This could be due to network latency, firewalls or other malfunctions at the LSS backend.

It is recommended, that the service provider implements some sort of timeout functionality which will terminate the operation after a period, and display a helpful message to the user.

5 Implementation

In this section, it is described how the communication flows of the previous section can be implemented.

5.1 Supported platforms

Both the service provider and the LSS supplier may choose technology to fit their requirements. However, it is a requirement, that common and standardized technology is chosen to ensure that a wide range of user devices and platforms are supported.

Both the service provider and the LSS supplier MUST ensure support for a specific set of browsers and platforms.

This set is listed at Nets-DanID. [SUB]

5.2 Web Messaging API

The communication between the service provider and the LSS-supplier is implemented using the HTML5 Web Messaging API [WM], which is an event-based JavaScript (JS) API and is widely supported by most browsers and on most devices.

Within the messaging paradigm, all commands are executed by sending a message. This is done asynchronously – note, that all methods in the API are void.

Code snippets in this section are meant for illustration purposes alone, and should not be used directly. For running code, please consult the test-examples.

5.3 Initialization

As a result of this architecture, both the service provider and the LSS Client initialization JS code has the following structure:

```
function onMessage(e) {
    //authenticate and handle message
}

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

registerMessageHandler();
//Possibly do further initialization
```

Figure 5: Initialization

The onMessage method is the message handler, doing all handling of incoming messages. This method will be discussed below.

The message handler is registered via both window.addEventListener and window.attachEvent – the former being supported is most modern browser, and the latter in IE5-8. In order to support IE8, registration via attachEvent is required.

After registration, the event-handler (onMessage) will be executed whenever the document receives a message from other documents [WM].

5.4 Message handling

The structure of the message handler is:

```
function onMessage(e) {
    var event = e || event;
    if (isMessageAuthentic(event)) {
        var message = JSON.parse(event.data);
        handleMessage(message);
    } else {
        //Log unauthentic message information
    }
}
```

Figure 6: OnMessage

On IE8, the occurred event is not passed as argument to onMessage, therefore the *var event = …* statement is required to handle both modern browsers and IE8.

Before handling any message, the message must be authenticated. This occurs in the isMessageAuthentic method and is described in Section 5.6.

All messages consist of the command name, e.g. *LssClientReady* and a content value defined by a JSON object mapping, containing parameter name/value-pairs.

The message handling proceeds in method handleMessage as follows:

```
function handleMessage(message) {
    if (message.command === "SomeCommand") {
        var content = JSON.parse(message.content);
        executeSomeCommand(content);
    }
    if (message.command === "OtherCommand") {
        //Execute other command
    }
}
```

Figure 7: HandleMessage method

The message.command is inspected for a supported command name, and the content is interpreted as a JSON structure, which is then handled appropriately.

5.5 Sending messages

When either party needs to send a message to the other, it is done as follows:

```
function sendMessage(command, content) {
    var message = {};
    message.command = command;
    message.content = content;
    var recipient = getRecipient();
    var recipientLocation = getRecipientLocation();
    recipient.postMessage(
              JSON.stringify(message),
              recipientLocation
    );
}
```

Figure 8: Send message

Here the content should be a JSON formatted string, such as

{ "KEY1" : "value1", "SECOND_KEY" : "val" }

and the getRecipient() and getRecipientLocation() methods have different implementations in the service provider page and the LSS Client:

```
function getRecipient() {
    //LSS Client implementation:
    return parent;
    //Service provider implementation:
    return document.getElementById("moceslss_iframe").contentWindow;
}
function getRecipientLocation() {
    //LSS Client implementation:
    return "*";
    //Service provider implementation:
    return "https://lss-for-nemid-server.dk";
}
```

Thus, the LSS Client will send messages to its *parent* whereas the service provider page needs to identify the *iframe* DOM element as recipient.

5.6 Message authentication

To ensure that only messages from the expected source are received and processed, an authentication mechanism based on an origin check is made on the received messages.

This is done by means of the following code:

```
function isMessageAuthentic(event) {
    //Service provider implementation:
    return event.origin == "https://lss-for-nemid-server.dk";
    //LSS Client pseudo-implementation:
    var message = JSON.parse(event.data);
    var content = JSON.parse(message.content);
    return event.origin == content.ORIGIN;
}
```

Figure 9: Message authentication

Authenticating the LSS Client is simple: All messages must come from the LSS Client URL.

Authenticating the service provider is a bit more complicated, since it involves inspecting the message contents parameter ORIGIN, and comparing it to the event origin. See Section 6.2.2 for details. Note, that the implementation in Figure 9 does not handle case-insensitivity of parameter names correctly.

If the received event message is not sent from the configured address, it could mean that an attacker is attempting to inject code into the SP flow. In that case, the flow should be stopped. The service provider decides how this is displayed to the user.

Further, it is recommended to error-log the event server-side along with the erroneous origin URL.

5.7 Service provider implementation

The service provider's handleMessage implementation would be structured along the following lines:

```
function handleMessage(message) {
    if (message.command === "LssClientReady") {
        //Log or save LSS information from message.content.
        var parameters = document.getElementById("flowParams").innerHtml;
        sendMessage("BeginFlow", parameters);
    }
    if (message.command === "ReceiveResult") {
        var result = JSON.parse(message.content);
        //Handle error depending on result.STATUS
        document.postBackForm.response.value = result.SIGNATURE;
        document.postBackForm.submit();
    }
}
```

When receiving the LssClientReady command, the flow parameters are immediately sent to the LSS Client.

The parameters generated by the service provider are set up in a script tag on the service provider's page:

```
<script type="text/x-nemidlss" id="flowParams">
{
    "SP_CERT":"MIIFij..==",
    "CLIENTFLOW":"login",
    "TIMESTAMP":"MjAxNC0wMS0yOCAxNDowNDozOSswMTowMA==",
    "REQUESTISSUER":"VFUgRXhhbXBsZQ==",
    "LANGUAGE":"da",
    "PARAMS_DIGEST":"R35g6+zBu1eXkn+6GXdZFwL/nBnOas8TV3Ewceo8Iyc=",
    "DIGEST_SIGNATURE":"jW6GgqyhO..=="
```

Figure 10: Flow parameters

Since an unknown script-type is defined the browser will not process the script.

When the result of the flow-operation is received – command *ReceiveResult* – the XML-DSig message is inserted into a form and posted to the service provider backend for validation.

We refer to the TU example demo applications for complete running code.

5.8 LSS implementation

The LSS implementation is similar. However, when the message handler is registered, the initialization code finishes by sending the *LssClientReady,* to have the service provider start the flow:

```
...
registerMessageHandler();

var clientReady = '{
                "API_VERSION" : "1.1.0.0",
                "LSS_SUPPLIER_ID" : "LSS-demo-provider",
                "LSS_INSTALLATION_ID" : "Demo",
                "LSS_VERSION" : "9.2.0.0"
               }';
sendMessage("LssClientReady", clientReady);
```

The LSS *handleMessage* implementation only has a single command to handle:

```
function handleMessage(message) {
    if (message.command === "BeginFlow") {
        //Begin the signing or logon flow, communicate with LSS backend
    }
}
```

When the user has selected certificate and a signature has been created in the LSS backend, the LSS Client will send the result to the service provider by using the Web Messaging API as shown in Figure 8.

Again we refer to the examples for complete running code.

5.9 LSS Client presentation to user

The service provider should await the *LssClientReady* command and not show the LSS Client to the user prior to this. By receiving this command, the service provider is made aware that the user has access to an LSS backend.

6 API specification

The JavaScript messages in the API sent between the LSS supplier and service provider are passed on as a JSON object – see Figure 10 for an example.

Each parameter is defined as a name-value pair. Both name and value are strings. The value may or may not be encoded using Base64.

The ordering of the parameters within the JSON structure holds no significance.

Although parameter names are presented in the sections below in upper case, they must be interpreted as case-insensitive. Examples can be found in the supplied demo applications on how to handle the case-insensitivity.

All parameter values are represented as strings. They may be Base64 encoded. Encoding is specified explicitly in tables below.

A Base64 encoded value should be interpreted as either a string or binary, as specified in the *Type* column in tables below.

Strings are UTF-8 encoded throughout. Note, that special care must be taken, if UTF-8 encoding/decoding is done in JavaScript.

Some values are case-insensitive. In that case, this will be explicitly stated in the description of the parameter.

Parameters may be mandatory in all flows (M), mandatory in signing flows (MS), optional (O), or optional in any signing flow (OS). These values are specified in *Required* columns in tables below.

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

Current version of the API is *1.1.0.0.*

6.1 The LssClientReady *command*

The following parameters form the *LssClientReady* command:

Name	Required	Encoding	Type
API_VERSION	M	None	string
LSS_SUPPLIER_ID	M	base64	string
LSS_INSTALLATION_ID	M	base64	string
LSS_VERSION	M	None	string
PDF_SUPPORTED	O	None	string

6.1.1 API_VERSION

This parameter gives the version of the implemented API.

Service providers may use this to determine the version of the LSS supplier's implementation.

For this version of the API, the value must be *1.1.0.0.*

6.1.2 LSS_SUPPLIER_ID

This parameter uniquely identifies the LSS supplier.

Service providers may use this e.g. for logging purposes.

6.1.3 LSS_INSTALLATION_ID

This parameter identifies the LSS installation at hand. The LSS could e.g. supply the name of the users organization here.

Service providers may use this for logging purposes.

6.1.4 LSS_VERSION

This parameter gives the version of the LSS installation.

By inspecting this value, the service provider would be able to prevent usage of a specific version of the LSS, e.g. if a severe bug has been detected.

The value must take the form *x.y.z.v* where *x, y, z,* and *v* are integer numbers. Version numbers should be comparable.

6.1.5 PDF_SUPPORTED

This optional value specifies whether PDF-signing is supported by the LSS at hand.

The value should be either *true* (default) or *false* (case-insensitive).

6.2 The BeginFlow *command*

The following parameters form the *BeginFlow* command.

Name	Required	Encoding	Type
CLIENTFLOW	M	none	string
ORIGIN	O	base64	string
ADDITIONAL_PARAMS	O	base64	string
ADDITIONAL_PARAMS_CRITICAL	O	base64	string
LANGUAGE	O	none	string
TIMESTAMP	M	base64	string
REQUESTISSUER	M	base64	string
SIGN_PROPERTIES	O	none	string
SIGNTEXT	MS	base64	string or binary

SIGNTEXT_FORMAT	MS	none	string
SIGNTEXT_MONOSPACEFONT	OS	none	string
SIGNTEXT_TRANSFORMATION	OS	base64	string
SIGNTEXT_TRANSFORMATION_ID	OS	none	string
SP_CERT	M	base64	binary
PARAMS_DIGEST	M	base64	binary
DIGEST_SIGNATURE	M	base64	binary

6.2.1 CLIENTFLOW

This parameter defines the flow-type. Value must be either *login* or *sign.* Value is case-insensitive.

6.2.2 ORIGIN

This parameter gives the URL of the service provider domain. The URL must have the form *https://<hostname>:<port>*, where *<hostname>* gives the fully qualified domain name of the page.

The *<port>* part must only be defined, if the service provider service is not using standard port-numbers.

The URL must not contain any path info and must not have any trailing slashes.

Examples:

https://serviceprovider.dk - allowed

https://serviceprovider.dk:9443 - allowed

https://another.provider.dk:443 - not allowed

https://serviceprovider.dk/ - not allowed

https://serviceprovider.dk/logonpage - not allowed.

When authenticating the command, the LSS backend must compare the value of this parameter to the *event.origin* – see Section 5.6. If the values don't agree, the LSS must return error *APP001.*

Note, that the benefit of this authentication mechanism stems alone from the fact, that the parameters are signed.

6.2.3 ADDITIONAL_PARAMS

This parameter contains a semi-colon separated list of named values.

E.g.:

property1=value1;property2=value2;lastKey=value3

This allows LSS suppliers to support custom functionality not directly supported by the API.

The LSS should ignore parameters not known to it – unless marked critical, see the following section.

6.2.4 ADDITIONAL_PARAMS_CRITICAL

This parameter contains a semi-colon separated list of names from the ADDITIONAL_PARAMS name-value list.

E.g.:

property2;lastKey

All parameters in the list must match a parameter name in ADDITIONAL_PARAMS.

These parameters must be handled as critical by the LSS, i.e. the LSS must return error code "LSSADP001" if a critical parameter is either unknown or cannot be handled.

6.2.5 LANGUAGE

The parameter specifies the client language. Value must be either *da* or *en* (case insensitive) to specify Danish and English, respectively, as the client language.

If not provided, language is assumed to be Danish.

6.2.6 TIMESTAMP

This parameter expresses current time when generating parameters.

LSS implementation must reject messages older than 3 minutes.

The timestamp must be supplied as the number of milliseconds since 1970-01-01 00:00:00 or as a formatted string.

Examples:

2013-12-17 13:33:47+0100

1395819294069

The formatted string may be obtained by using the following format-strings:

yyyy-MM-dd HH:mm:ssZ - Java

yyyy-MM-dd HH:mm:sszzz - .Net

6.2.7 REQUESTISSUER

The service provider "Friendly Name" presented to users in logon flows.

This parameter will be part of the resulting XML-DSig message.

The LSS-implementation must show this value to the user in logon flows.

The LSS-implementation must display this value as clear-text and must carefully escape any dynamic content.

The service provider must use the value ("Friendly Name") agreed upon in their current service agreement with Nets DanID.

6.2.8 SIGN_PROPERTIES

The value of this parameter must be XML formatted XML-DSig SignatureProperty elements. The elements will be included in the resulting XML-DSig message.

We refer to the general service provider documentation [TU] for details on how to format this parameter.

6.2.9 SIGNTEXT

The actual text signed by the user in signing flows. The form of the text is given by the SIGNTEXT_FORMAT parameter – see below.

When SIGNTEXT_FORMAT *text, xml,* or *html* is specified the value must be interpreted as a string.

When SIGNTEXT_FORMAT *pdf* is specified the value must be interpreted as binary.

6.2.10 SIGNTEXT_FORMAT

This parameter specifies the format of the SIGNTEXT parameter.

The case-insensitive value must be one of

- *text*
- *html*
- *xml*
- *pdf*

6.2.11 SIGNTEXT_MONOSPACEFONT

When given value *true* (case-insensitive) this parameter indicates that plain text should be rendered with a mono-spaced font to allow for indention based formatting. Further, no word wrapping is allowed.

The parameter has no effect unless the SIGNTEXT_FORMAT is specified as *text.*

When this parameter is not present – or has any value other than *true* – plain text is rendered using the default font.

6.2.12 SIGNTEXT_TRANSFORMATION

The value of this parameter must be an XSLT style sheet. The style sheet is used to transform the XML passed in parameter SIGNTEXT into an HTML document, which is displayed to the user for signing.

This parameter is mandatory if SIGNTEXT_FORMAT is *xml.*

6.2.13 SIGNTEXT_TRANSFORMATION_ID

This parameter specified an optional identifying string for the style sheet defined in SIGNTEXT_TRANSFORMATION. The parameter will be part of the resulting XML-DSig message.

If a service provider uses several different style sheets, this identifier provide a means to assess which style sheet was applied – note that SIGNTEXT_TRANSFORMATION is *not* part of the resulting XML-DSig message.

The parameter has no meaning unless in an XML signing flow.

6.2.14 SP_CERT

This parameter gives a DER representation of the certificate used for signing the parameters.

The certificate must belong to the service provider and must be issued by a Nets-DanID trusted Certificate Authority.

The LSS implementation must validate this certificate to make sure, that the *BeginFlow* command is authentic.

6.2.15 PARAMS_DIGEST

The parameter value is a SHA-256 digest of the normalized parameters.

See section 7 for detailed documentation of the normalization and signing procedure.

6.2.16 DIGEST_SIGNATURE

This parameter gives an RSA-SHA256 signature computed on the PARAMS_DIGEST.

See section 7 for detailed documentation of the normalization and signing procedure.

6.3 *The* ReceiveResult *command*

This section specified parameters used in the *ReceiveResult* command.

Name	Required	Encoding	Type
STATUS	M	none	string
STATUS_TEXT	O	base64	string
SIGNATURE	O	base64	string

6.3.1 STATUS

The parameter gives the overall status of the sign/logon operation.

See Section 9 for possible status codes.

6.3.2 STATUS_TEXT

When defined, this parameter gives a detailed technical description of the error that occurred.

The text may be used for logging and further error handling by the service provider. The service provider may assume that the value is clear-text without any special formatting.

Due to its technical nature, this parameter value should never be displayed to users.

Note, that this parameter is mandatory if STATUS is *LSSERR001.*

6.3.3 SIGNATURE

If the sign/logon flow is successful, this parameter contains the XML-DSig message signed by the user.

This parameter is mandatory if STATUS is *LSS000.*

The XML-DSig message must use the same format as existing NemID variants and as a minimum OOAPI must be able to validate the message and extract the required information from it.

A detailed specification of the XML-DSig message is given in the document "LSS Technical specification".

This means, that the service provider could (and should) handle the XML-DSig message in the exact same way, regardless of the NemID variant chosen by the user.

Validation of the XML-DSig message could be done using the OOAPI package – or other available validation implementations - and is described in DanID's service provider package. [TU]

7 Parameter validation

The LSS-implementation must validate all received parameters.

New in version 2.0 is the requirement, that the implementation must ignore unknown parameters silently. The signature validation must be processed as before, but new or unknown parameters should not cause errors.

The service provider is required to supply a valid signature on the parameters, along with their certificate, as described in the following section.

This signature must be validated by the LSS-implementation. The LSS must verify that

- the parameter signature is valid
- the signing certificate is valid temporally
- the signing certificate is not revoked
- The signing certificate is issued by a trusted party – usually DanID.

The mandatory TIMESTAMP parameter (Section 6.2.6) must be validated, and no requests older than three minutes should be accepted.

Validating the signature and the time stamp enables the LSS-supplier to log a cryptographic proof that the given service provider requested the specific service at the given time.

7.1 Parameter signature

To ensure the integrity of the parameters in transit between the service provider and the LSS backend, they must be signed by the service provider.

The process for signing the parameters is:

1. The service provider collects the list of parameters. The list is normalized (see Section 7.2) into a string, and the SHA-256 digest value of the string's UTF-8 representation is calculated.

2. The normalized string is signed. The signature is performed using the VOCES certificate, which is associated with the service provider's 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, respectively.

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.

7.2 Parameter normalization

The digest of the LSS Client parameters is calculated on 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. This means that all characters be converted to lowercase before sorting.

2. Each parameter is concatenated to the result string as an alternating sequence of name and value: $name_1$ || $value_1$ || $name_2$ || $value_2$ || … || $name_n$ || $value_n$. The names must be as passed to the LSS and not in the lowercase representation used in the first step.

3. Finally, the concatenated string is encoded using UTF-8, and the digest is computed on the resulting bytes.

8 Signing PDF documents

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 for general documentation on the supported signing types and validation.

The exception is the support for the PDF signing flow. In the current version, the LSS backend support for this is optional and is flagged in the PDF_SUPPORTED parameter.

If a signing operation is unsuccessful, an error code is returned to the service provider. The error code is provided to the service provider base64 encoded. Section **Error! Reference source not found.** contains a list of the error codes, which a service provider may receive from a signing operation.

9 Status codes

If the flow is completed successfully, the status code LSS000 is returned.

In the event of an error, a status code is received indicating what went wrong. In addition to the status code, an optional text may be returned to the service provider in the STATUS_TEXT parameter.

9.1 General status codes

These status codes are general to the client functionality and may be received, regardless of which operation the client was supposed to do.

Status 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.
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.

9.2 Signing status codes

Status 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.

9.3 LSS specific status codes

The following codes may be returned during LSS operations.

Status code	Status code interpretation
LSS000	This code is used to signal success.
LSSERR001	When this code is returned an unspecified error occurred.
LSSPDF001	The LSS back-end does not support PDF signing.
LSSAUTH001	The user is not able to authenticate.
LSSSRV001	The signature on the client parameters could not be verified.
LSSGLB001	This error must not be returned by any LSS-supplier, but is reserved for use from the global LSS Client.
LSSJSN001	Error parsing JSON object
LSSADP001	ADDITIONAL_PARAMS not supported. Is used when the LSS receives an ADDITIONAL_PARAMS parameter containing one or more unsupported keys or values which is marked as critical in "ADDITIONAL_PARAMS_CRITICAL".

9.4 Status code handling

All the general status codes are shared with the general service provider package and should be handled in the same manner as in that context.

It is required that the LSS back-end communicates success or error through one of the defined status codes to the service provider through the LSS Client. All errors occurring at the LSS back-end will be communicated through the API.

Unhandled JavaScript errors might cause the flow and communication to stop. The LSS back-end is required to ensure that JavaScript errors are caught and a status code is sent to the service provider if unable to complete the flow. Simple try-catch clauses could provide a similar mechanism for the service provider but has been omitted from the examples in order to simplify the example code.

It is demonstrated in the code examples found in the LSS for NemID SP package how to send and receive the status codes. The service provider examples provide a mechanism to convert error codes into text messages explaining the user of the problem.

It is considered best practice to populate a form with the status code along with other received parameters in JavaScript when received through the Web Messaging API and then post these to a web-page handling the success/error scenario as demonstrated in the demo examples.

The following table specifies how the service provider should handle the LSS specific status codes

Status code	**Service provider action**
LSS000	Proceed to normal flow handling the result
LSSERR001	The user encountered a problem with the LSS flow. The error is unknown to the service provider.
LSSPDF001	The LSS backend does not support PDF signing. The service provider must offer the user to sign in a different format
LSSAUTH001	The user encountered a problem with the LSS flow, but has been notified about errors inside the LSS iframe.
LSSSRV001	The service provider should check if the used VOCES is still valid. If it is valid the service provider should review his code generating the signature on the parameters
LSSGLB001	The service provider should tell the user to connect to his enterprise LAN as this status code indicates that he is not connected
LSSJSN001	The status code indicates that the service provider has a programming error which he should fix
LSSADP001	The code indicates that the LSS backend does not understand the additional parameters given. The service provider should terminate the flow telling the user that his organization is non-compliant with the flow

10 Issuing certificates

As of version 1.1.0.0 of the API specification LSS implementations must support issuing of certificates.

The issuance page at Nets DanID will attempt to detect if a LSS implementation is present on the user network. If not the process will proceed as usual. If a LSS implementation is detected and the API version equal or above 1.1.0.0 the employee will be redirected to the LSS issuance page.

LSS implementations must respond to the following URLs

- https://lss-for-nemid-server.dk/enroll/
- https://lss-for-nemid-server.dk/enroll/<cvr>/<reference-number>

where CVR refers to the company registration number of the employee in question. Reference number refers to the reference number given by Nets DanID, when the LRA orders a new NemID for business. If the first URL variant is used the employee must by prompted for those values

On the LSS issuance page the employee must be prompted for the issuance PIN provided by Nets DanID by either mail or through the LRA interface (immediate issuance)

The LSS vendor may prompt the user for any other additional information, necessary for associating the certificate with an employee, during the issuance process.

11 Recommended security reading

Following a list of URL's which provide a recommended read during development and integration with the LSS Client and LSS back-end. The info is relevant for both service providers and LSS suppliers.

OWASP HTML5 Security Cheat Sheet:

https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

12 References

[TU]	DanID TU-pakke https://www.nets-danid.dk/tu-pakke
[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
[SUB]	Description of internet browsers and OS platforms that must be supported. https://www.nets-danid.dk/kundeservice/krav_til_computer/internetbrowser/index.html
[WM]	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