HTML5 and Digital Signatures

Signature Creation Service 1.0.1

June 30, 2015

Population Register Centre

DOCUMENT MANAGEMENT

Prepared by	Pekka Laitinen / VRK <pekka.laitinen@vrk.fi>
Inspected by	
Approved by	

VERSION CONTROL

version no.	what has been done	date/person
0.1	Initial draft	2.9.2014/PL
0.2	Initial draft version ready	5.9.2014/PL
0.3	New version based on comments	8.9.2014/PL
0.4	Draft status, minor fixes	12.9.2014/PL
0.5	Added • version check functionality • signature creation request with GET method • signing of plain hashes • other minor fixes • grammar/sanity check	6.10.2014/PL
0.6	Atrribute renaming	9.10.2014/PL
0.7	Minor fixes on attribute renamings, added more details on error handling, and appended security considerations section.	13.10.2014/PL
1,0	Final version (no changes to 0.7 version)	17.2.2015/PL
1.0.1	Added • https usage as requirement • server certificate requirements • SCS module for javascript (example) • clarifications for selector.akis and selector.issuers format • removed selector.validate parameter • added requirement for SCS to check whether certificate is valid or not • removed localhost.fineid.fi as dnsName parameter in SCS server certificate (compared to previous 1.0.1 version) • protocol version is still the same: 1.0	30.6.2015/PL

Table of contents

HTML5 AND DIGITAL SIGNATURES

1 Introduction

This specification describes a method to generate digital signatures in HTML5 applications [HTML5] that are executed in User Agents, i.e., web browsers. The specification utilizes the Cross-Origin Resource Sharing (CORS) specification [CORS] that enables an HTML5 application downloaded from Site A to communicate with a service located in Site B using Javascript's XMLHttpRequest mechanisms [XHR], for instance. The communication protocol uses HTTP protocol and the information elements are transferred using JSON format [JSON].

The HTML5 application makes a signature request by sending the data that needs to be signed to the Signature Creation Service (SCS). Upon receiving the request, the SCS displays a certificate selection dialog to the end user, who will select the certificate that will be used to generate the digital signature. If required, the end user enters the PIN code for accessing the private key to generate the signature. Once the signature is created, the SCS sends the signature along with the certificate chain and other needed information to the HTML5 application. Upon receiving the digital signature, the HTML5 application uses it according to its specifications.

1.1 Definitions and Acronyms

CORS	Cross-Origin Resource Sharing
CRL	Certificate Revocation List
JSON	JavaScript Object Notation
OCSP	Online Certificate Status Protocol
OS	Operating System
SCS	Signature Creation Service

1.2 References

[CORS]	Cross-Origin Resource Sharing, W3C Recommendation, January 16, 2014, http://www.w3.org/TR/cors/
[HTML5]	HTML5, W3C Candidate Recommendation, July 31, 2014, http://www.w3.org/TR/html5/
[HTTP]	Hypertext Transfer Protocol -- HTTP/1.1, June 1999, https://www.ietf.org/rfc/rfc2616.txt
[JSON]	JavaScript Object Notation Introduction, http://json.org
[XHR]	XMLHttpRequest Level 1, W3C Working Draft, January 30, 2014, http://www.w3.org/TR/XMLHttpRequest/

[PKIX] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, May 2008
https://www.ietf.org/rfc/rfc5280.txt

[HTTPS] HTTP over TLS, May 2000,
https://www.ietf.org/rfc/rfc2818.txt

2 Signature Creation Service (SCS)

The SCS functions as a simple web server that provides access to signature creation functionality via HTTP/1.1 protocol [HTTP].

2.1 Requirements

The SCS SHALL support the following functionality:

- The SCS SHALL support https scheme and it MAY support http scheme for browser access. The SCS SHALL generate a key pair, generate a server certificate and import that certificate to device’s trusted certificate store. Additional requirements can be found in chapter 2.3.
- The SCS SHALL support CORS [CORS]:
  - Each HTTP response SHALL contain the `Access-Control-Allow-Origin` header with value "*".
  - When User Agent makes the CORS preflight request, i.e., the OPTIONS request, the corresponding HTTP response SHALL contain the following headers: `Access-Control-Max-Age` header with default value "`3600`", `Access-Control-Allow-Methods` header with default value "`GET, POST`", and `Access-Control-Allow-Headers` headers with default value "`Content-Type, Accept`".
- The SCS SHALL process all HTTP requests.
- The SCS SHALL show certificate selection dialog for every signature creation request it receives.
- The SCS SHALL process HTTP requests where the `Origin` header containing `https` protocol in the URL. HTTP requests with Origin header containing `http` protocol in the URL SHALL not be processed.
- The SCS SHALL show the content of the `Origin` header of the signature creation request to the end user (so that end user is able to identify the origin of the signature creation request). It MAY be included in the certificate selection dialog.
- The SCS SHALL have only one active signature creation request active at a time. If another signature creation request is received, the SCS shall silently ignore the request and send a response to the requesting HTML5 application that another request is already active.
- The SCS SHALL respond to all incoming requests.

- The SCS SHALL be able to handle HTTP requests where content length is up to 2MB (=2*1024*1024 bytes). The SCS MAY be able to support larger HTTP requests.

- The client application SHALL be able to send the data that is to be signed for signing, i.e., the SCS will receive the full to-be-signed data, and calculate the digest of this data itself.

- The client application SHALL be able to send the digest of the data that is to be signed for signing, i.e., the client application will calculate the digest of the to-be-signed data, and send it to the SCS.

- The SCS MAY validate each end entity certificate (that is available on the device) and its certificate chain. If SCS validates end entity certificates and finds out that one expired, this end entity certificate is not shown to the end user as a selection option.

  NOTE: Even if the SCS validates end entity certificates, the server consuming digital signatures generated by SCS must also validate the end entity certificate and its corresponding certificate chain when checking the signature.

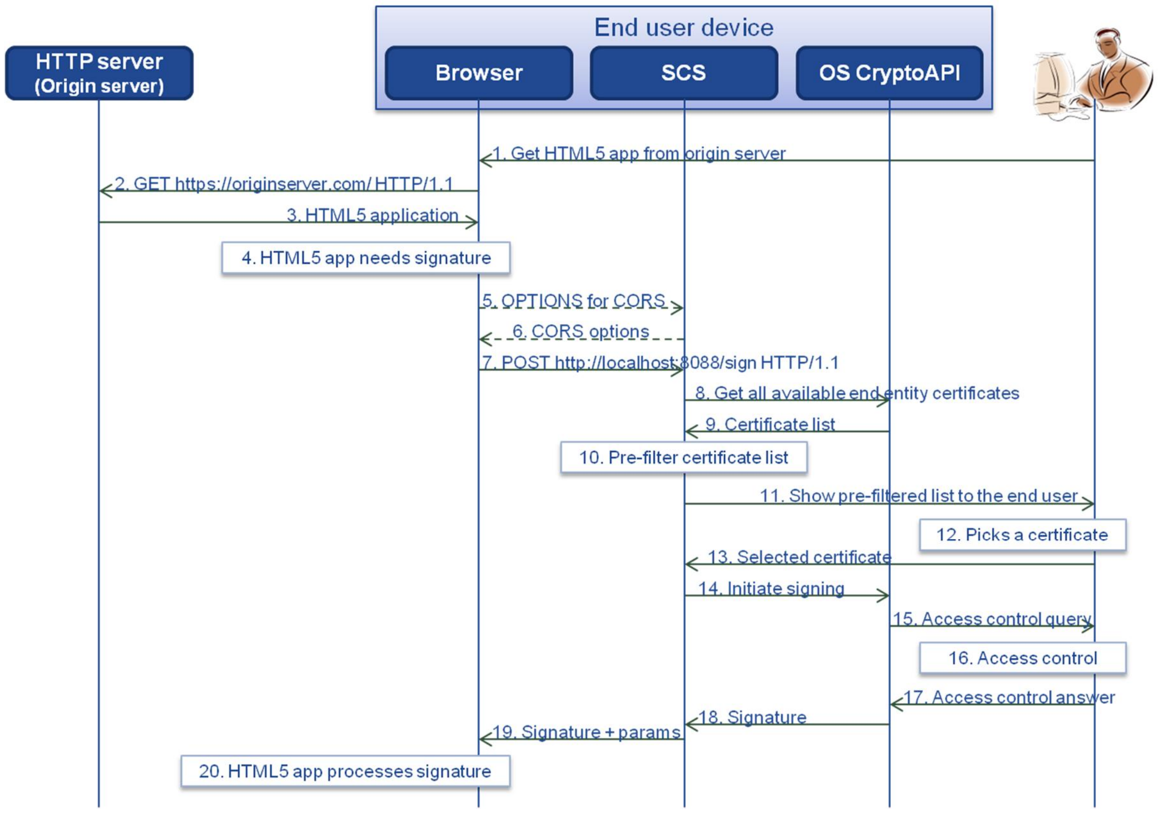
- The SCS MAY pre-filter applicable certificate list (before they are shown to the end user) and certificate filtering MAY be based on one or more values in end user certificates:

  - Issuer DN field,
  - KeyUsage extension values, and/or
  - AuthorityKeyIdentifier extension.

  This way the HTML5 application is able to indicate to the SCS the acceptable end user certificates.

2.2 Message sequence diagram

Below is an example message sequence diagram of signature generation for HTML5 applications and SCS functionality. This specification concentrates on the interface between the browser and the SCS, namely steps 5, 6, 7, and 19. Moreover, the steps 5 and 6 are specified by the CORS specification. Steps from 8 to 18 are described for completeness but the actual steps depend on the OS architecture.


End user device
HTTP server (Origin server)
Browser
SCS
OS CryptoAPI
1. Get HTML5 app from origin server
2. GET https://originserver.com/ HTTP/1.1
3. HTML5 application
4. HTML5 app needs signature
5. OPTIONS for CORS
6. CORS options
7. POST http://localhost:8088/sign HTTP/1.1
8. Get all available end entity certificates
9. Certificate list
10. Pre-filter certificate list
11. Show pre-filtered list to the end user
12. Picks a certificate
13. Selected certificate
14. Initiate signing
15. Access control query
16. Access control
17. Access control answer
18. Signature
19. Signature + params
20. HTML5 app processes signature

1. The end user types the URL of the origin server to the browser.
2. The browser makes a request to the URL. It is recommended that the URL uses https scheme.
3. The browser downloads the HTML5 application, which typically consists of several HTTP requests being made by the browser to the origin server. If the HTML5 application is downloaded using secure connection, i.e., https scheme, the HTTP requests to the SCS in step 5 and step 7 must be done over https scheme as well. If the HTML5 application is downloaded using unsecure connection, i.e., plain http scheme, the requests to the SCS in step 5 and step 7 must be done over http scheme as well.
4. At one point, application logic of the HTML5 application requires that a digital signature is needed.
5. The browser makes an OPTIONS request to the SCS to discover whether CORS type requests are allowed. This request may been done also earlier.
6. The SCS returns the CORS options allowing requests being made from the current HTML5 application context (identified by Origin).
7. The HTML5 application makes a signature creation request.
8. The SCS queries local crypto API for all available end entity certificates.
9. The Crypto API returns all available end entity certificates.

10. The SCS pre-filters the end entity certificate list if signature creation request contained filters (selectors based on issuer name, authority key identifier or required key usage parameters). Additionally, the SCS may locally validate the end entity certificates, and filter out the ones that are not valid.
11. The SCS presents the end user with a list of (pre-filtered) end entity certificate list.
12. The end user selects one of the listed certificates.
13. The selected certificate is returned to the SCS.
14. The SCS initiates the digital signature generation with the private key associated with the selected certificate and with given signature parameters received in the signature creation request.
15. Crypto API requests end user to give authorization to the usage of the selected private key.
16. End user enters the authorization data (e.g., PIN code).
17. Authorization data is returned to the Crypto API. If it was correct, the crypto API generates the signature.
18. Signature is returned to the SCS.
19. The SCS generates the signature creation response (with the signature, signature algorithm, and the certificate chain of the selected end entity certificate) and returns it to the HTML5 application.
20. The HTML5 application processes the signature including validating the certificate chain of the used end entity certificate (e.g., validity and revocation status). The validation can also be done by the origin server (not depicted here).

Variations possibilities:

- Before showing the end user the certificate list for certificate selection, the SCS could show a notification what is being signed, i.e., show the actual data that is being signed. However, if data is not textual (e.g., XML signature element, or plain binary data), the end user might get confused by this notification.

2.3 Server certificate

In order for SCS to use https based scheme to communicate with browsers, it must generate a local key pair and a server certificate that needs to be imported to the devices trusted certificate store. The browser that wishes to use SCS must then use that trusted certificate store when validating server certificates.

Upon installation of the SCS to the target device, it SHALL generate two key pairs, one for the local root certificate that is going to be used sign the server certificate. After the certificates have been generated, the SCS SHALL delete the key pair of the root certificate, and store the key pair of the server certificate securely. The certificate chain, i.e., the root certificate and the server certificate SHALL be imported to the trusted certificate store of the target environment, or any other certificate store that the web browsers in the device are using.

The key pairs and the certificates have the following requirements:

- The key pairs SHALL be at least 2048 bits in the case of RSA key pair.
- The server certificate SHALL have the following attributes:
    - The SubjectDN parameter SHALL include CommonName attribute containing “127.0.0.1”.
    - The certificate SHALL contain SubjectAltName extension with at least following attributes:
        - extension is not critical,
        - dNSName: “localhost”, and
        - iPAddress: “127.0.0.1”.
    - The certificate SHALL contain KeyUsage extension with following attributes:
        - extension is critical,
        - digitalSignature key usage, and
        - keyEncipherment key usage.
    - The certificate SHALL contain ExtendedKeyUsage extension with at least following attributes:
        - extension is not critical, and
        - serverAuth key usage.
- Additionally, the certificate SHALL include any attributes that are specified in [PKIX] and [HTTPS].
- The SCS SHALL generate a self-signed root certificate from which the server certificate is issued. The root certificate SHALL contain attributes as specified in [PKIX].

2.4 CORS preflight check

A browser that supports CORS will do a preflight check to see if a CORS request is allowed to be sent to the SCS. This is done by sending an OPTIONS request to the web server indicating the Origin of the requesting HTML5 application, and the allowed request methods and headers. An example of such request is below.

```
OPTIONS /sign HTTP/1.1
Host: localhost:8088
Connection: keep-alive
Access-Control-Request-Method: POST
Origin: https://vrk.fineid.fi
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/36.0.1985.143 Safari/537.36
```

```
Access-Control-Request-Headers: accept, content-type
Accept: */*
Referer: https://vrk.fineid.fi/scstest/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: fi-FI,fi;q=0.8,en-US;q=0.6,en;q=0.4,fr;q=0.2
```

As a response, the SCS should send an HTTP response back with specified CORS headers. An example of such a response is below.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Server: VRKLocalhostSigner 0.7.1 beta (2014-09-08)
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Accept, Content-Type
Access-Control-Max-Age: 3600
Access-Control-Allow-Origin: *
Accept: application/json, */*
Date: Wed, 08 Sep 2014 07:38:01 GMT
```

2.5 Version check

2.5.1 Request

The HTML5 application application checks the version of the SCS as well as discover the supported mechanisms, and availability of the SCS by sending an XMLHttpRequest to the SCS.

HTTP parameters:

- HTTP-Method: GET
- Request-URI: /version
- Other HTTP parameters are filled in by the browser as specified by HTTP/1.1 and CORS specifications.

Example HTTP request for version check.

```
GET /version HTTP/1.1
Host: localhost:8088
Connection: keep-alive
Accept: application/json, text/javascript, */*; q=0.01
Origin: https://vrk.fineid.fi
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/36.0.1985.143 Safari/537.36
Referer: https://vrk.fineid.fi/scstest/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: fi-FI,fi;q=0.8,en-US;q=0.6,en;q=0.4,fr;q=0.2
```

2.5.2 Response

The SCS responses to the version check request by sending version check response.

HTTP protocol parameters:

- Status-Code: 200
- Content-Type: application/json
- Other HTTP parameters are filled in by the browser as specified by HTTP/1.1 and CORS specifications.

Version check response:

- **version** (string, mandatory): the version of the specification the SCS supports. For this specification the value is "1.0".
- **httpMethods** (string, mandatory): the allowed HTTP methods for signature creation request. The value for this specification is "GET, POST".
- **contentTypes** (string, mandatory): the supported data types supported by the SCS. The value for this specification is "data, digest".
- **signatureTypes** (string, mandatory): the supported signature types supported by the SCS. The value for this specification is "signature".
- **selectorAvailable** (boolean, mandatory): determines whether the SCS supports the selector functionality. Possible values are "true" or "false".
- **hashAlgorithms** (string, mandatory): the supported hash algorithms supported by the SCS. The value for this specification is "SHA1, SHA256, SHA384, SHA512".

The version check can be used by the client application to discover the service, and validate that it is up and running.

```
HTTP/1.1 200 OK
Content-Length: 24
Content-Type: application/json
Server: VRKLocalhostSigner 0.7.1 beta (2014-09-08)
Access-Control-Allow-Origin: *
Accept: application/json, */*
Date: Wed, 08 Sep 2014 07:35:16 GMT

{
  "version": "1.0",
  "httpMethods": "GET, POST",
  "contentTypes": "data, digest",
  "signatureTypes": "signature",
  "selectorAvailable": true,
  "hashAlgorithms": "SHA1, SHA256, SHA384, SHA512"
}
```

2.6 Signature creation

2.6.1 Request (POST)

The HTML5 application makes a signature creation request by sending an XMLHttpRequest to the SCS.

HTTP parameters:

- HTTP-Method: POST
- Request-URI: /sign
- Content-Type: application/json
- Other HTTP parameters are filled in by the browser as specified by HTTP/1.1 and CORS specifications.

Signature creation request parameters:

- **version** (string, optional) contains the version of the SCS specification version that the HTML5 application expects the SCS to support (which for this version is "1.0").
- **selector** (object, optional) contains the certificate selector parameters
  - **issuers** (array, optional) a list of acceptable issuers of the end user certificate in string format (e.g., "CN=Trusted CA, OU=Unit, O=Organisation, C=FI") or in ASN1/DER encoded form as it is present in the end user certificate (i.e., the issuer field in the end user certificate [PKIX]) in which case the format is “base64:<ASN1/DER encoded issuer field as it is present in the end user certificate in base64 encoded format>”. Values are case sensitive. If specified, the end user certificate MUST have one the listed issuers as the issuer.

    It is recommended to pre-filter the certificate list using the “akis” array field instead of the “issuers” array field.
  - **akis** (array, optional) a list of authority key identifiers of acceptable issuers in base64 format (key identifier is the content of the “keyIdentifier” field in the AuthorityKeyIdentifier extension in the end user certificate as specified in chapter 4.2.1.1 [PKIX], i.e., either the SHA-1 hash of the authority certificate’s public key or 60 least significant bits of this hash preceded by ‘0100’ bits as specified in chapter 4.2.1.2 in [PKIX]). Values are case sensitive. If specified, the end user certificate MUST have one of the listed keyIdentifiers present in the AuthorityKeyIdentifier extension.
  - **keyusages** (array, optional) a list of required keyusages as they are listed in KeyUsage extension ("digitalSignature", "nonRepudation", "dataEncipherment", "decipherOnly", "encipherOnly", "keyAgreement", "keyEncipherment", "keyCertSign", "crlSign"). Values are case insensitive. If specified, the end user certificate MUST have all the listed key usages present in the KeyUsage extension.

- **content** (string, mandatory) contains the data to be signed that is base64 encoded.
- **contentType** (string, optional) specifies the type of the data field. Default is "data". Options for type are "data" meaning that the data field contains the data that should be signed, and "digest" meaning that the data field contains the digest of the data that should be signed. Supported values are "data" and "digest".
- **hashAlgorithm** (string, optional) specifies the requested signature algorithm. Default is "SHA256" indicating "SHA256withRSA" signature with RSA keys, and "SHA256withECDSA" signature with EC keys. Supported values are "SHA1", "SHA256", "SHA384", and "SHA512". Note that if the contentType parameter value is "digest", the data parameter must contain a digest calculated using the digest algorithm indicated in the algorithm parameter.
- **signatureType** (string, optional) specifies the requested signature format. Default is "signature" indicating plain signature (RSASSA-PKCS1-V1_5 for RSA keys and ECDSA for EC keys). Supported values are "signature".

Example HTTP POST request of signature creation request:

```
POST /sign HTTP/1.1
Host: localhost:8088
Connection: keep-alive
Content-Length: 155
Accept: application/json, text/javascript, */*; q=0.01
Origin: https://vrk.fineid.fi
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/36.0.1985.143 Safari/537.36
Content-Type: application/json
Referer: https://vrk.fineid.fi/scstest/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: fi-FI,fi;q=0.8,en-US;q=0.6,en;q=0.4,fr;q=0.2

{
  "version": "1.0",
  "selector": {
    "issuers: [
      "CN=VRK CA for Qualified Certificates, ..., C=FI",
      "CN=VRK CA for Qualified Certificates - G2, ..., C=FI"
    ],
    "keyusages": [
      "nonrepudiation"
    ]
  },
  "content": "VGhpcyBpcyB0aGUgZGF0YSB0byBiZSBzaWduZWQuLi4=",
  "contentType": "data",
  "hashAlgorithm": "SHA1",
  "signatureType": "signature"
}
```

2.6.2 Request (GET)

The HTML5 application may send the signature creation request also using GET method. The HTTP protocol and request parameters are the same as with POST method with following exceptions:

HTTP parameters:

- Request-Method: GET

Signature creation request parameters:

- **selector** functionality is not supported with GET method.

Example HTTP GET request of signature creation request:

```
GET /sign?content=VGhpc...QuLi4=&hashAlgorithm=SHA1&contentType=digest
HTTP/1.1
Host: localhost:8088
Connection: keep-alive
Accept: application/json, text/javascript, */*; q=0.01
Origin: https://vrk.fineid.fi
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/36.0.1985.143 Safari/537.36
Referer: https://vrk.fineid.fi/scstest/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: fi-FI,fi;q=0.8,en-US;q=0.6,en;q=0.4,fr;q=0.2
```

2.6.3 Response

The SCS responses to the signature creation request by sending the signature creation response.

HTTP protocol parameters:

- Status-Code: 200
- Content-Type: application/json
- Other HTTP parameters are filled in as specified by HTTP/1.1 and CORS specifications.

Signature creation response:

- **version** (string, mandatory) contains the version of the SCS specification version that the SCS supports (which for this version is "1.0").
- **status** (string, mandatory) indicates whether the signature creation operation was successful.
    - "ok" indicates that the operation was successful,
    - "failed" indicates that the operation failed.

- **reasonCode** (integer, mandatory) gives the reason code of the response status. Possible values are listed in section 2.7.
- **reasonText** (string, mandatory) gives textual description of the reason code.
- **signature** (string, optional) contains the digital signature that is base64 encoded. Present if signature creation operation was successful.
- **signatureType** (string, optional) contains the type of the signature. Supported values are "signature". Present if signature creation operation was successful.
- **signatureAlgorithm** (string, optional) contains the format of the digital signature. Supported values are: "SHA1withRSA", "SHA256withRSA", "SHA384withRSA", "SHA512withRSA", "SHA1withECDSA", "SHA256withECDSA", "SHA384withECDSA", and "SHA512withECDSA". Present if signature creation operation was successful.
- **chain** (array, optional) contains the certificate chain of the end user certificate. The end user certificate is at index 0. All certificates are base64 encoded. Present if signature creation operation was successful. The chain may be incomplete and contain only the end user certificate.

Example HTTP response of signature creation response:

```
HTTP/1.1 200 OK
Content-Length: 5858
Content-Type: application/json
Server: VRKLocalhostSigner 0.7.1 beta (2014-09-08)
Access-Control-Allow-Origin: *
Accept: application/json, */*
Date: Wed, 08 Sep 2014 07:38:16 GMT

{
  "version": "1.0",
  "signatureAlgorithm": "SHA256withRSA",
  "signatureType": "signature",
  "signature":
    "eHF1oXKD62+KTSUb3GPNMhYumwjtL...EVvvo++k+fzDK41nvK5lAZSkA==",
   "chain":[
      "MIIGADCCBOigAwIBAgIEBfetjDA...iLniyk0+r5aasoF7TY9n4tpMWmg==",
      "MIIFcDCCBFigAwIBAgIDAdbDMA0...fBVhROQzKoDAk1DPtg1gOADYfLg==",
      "MIIEHjCCAwagAwIBAgIDAdTAMA0...lEtXfFCfv9DA8MeVIUfa2pTH6Tk="
   ],
  "status": "ok",
  "reasonCode": 200,
  "reasonText":"Signature generated"
}
```

2.7 Reason codes

The reason codes for signature creation response messages are modeled after the HTTP status codes:

- 2xx series indicates a successful operation.

- 4xx series indicates that the operation was cancelled due to some reason, and
- 5xx series indicates that the SCS is malfunctioning or not supporting requested functionality.

The reason codes define general categorization of the operation result (e.g., ok, user cancelled, bad request), and are not intended to give detailed description to the HTML5 application. More detailed error handling should be done internally by the SCS.

The reason code related response parameters:

- **status** (string, mandatory): "ok" if the operation was successful, "failed" if the operation was unsuccessful. If the reasonCode value is 200, then status is "ok". With any other reasonCode value, the status is "failed".
- **reasonCode** (integer, mandatory): possible reason code values are listed on the table below. This field intended for general categorization of the error class, and should be used generate end user understandable error message in desired language.
- **reasonText** (string, mandatory): the textual error description. It is recommended that description takes the form "<Text>: <Description>" where <Text> is the text in the Text column in the table below, and the <Description> contains more detailed description of the error. This field is mainly intended for developers, and should not be shown to the end user.

Code	Text	Explanation
200	OK	Signature was created successfully
400	Bad request	Request was malformed, e.g., - unknown Request-URI - bad JSON format - missing mandatory parameters in signing request - a parameter contains a value that is not supported
401	Unauthorized	End user cancelled operation, e.g.: - end user did not select a certificate, - usage of the key was not authorized (e.g., PIN mistyped, PIN typing cancelled, or PIN locked) - end user does not have any applicable certificates
403	Forbidden	Request was denied, e.g.: - another request is being processed already - request was made from unidentified origin, i.e., https protocol was not used to download HTML5 app requesting the signature
413	Request Entity Too Large	Request size was too big
500	Internal Server Error	SCS is not working properly, e.g., is misconfigured
501	Not Implemented	Requested functionality is not implemented.

An example error response:

```
HTTP/1.1 200 OK
```

```
Content-Length: 88
Content-Type: application/json
Server: VRKLocalhostSigner 0.7.1 beta (2014-09-08)
Access-Control-Allow-Origin: *
Accept: application/json, */*
Date: Wed, 08 Sep 2014 07:38:16 GMT

{
  "status": "failed",
  "reasonCode": 401,
  "reasonText":"Not Authorized: end user cancelled operation"
}
```

3 Security considerations

This section is non-normative.

3.1 Authentication

The SCS can be used for end user authentication as well. For instance, a server may create a challenge, which is then digitally signed by the SCS using an end user certificate intended for authentication purposes. The authentication is validated by the server, it the signature is valid for the given challenge. The challenge should contain nonce and timestamp values, which the server should check to be valid to prevent replay attacks.

Additionally, the SCS could include a client nonce to the challenge so that the SCS would generate a different signature even if the challenge remains the same (similarly as described in section 3.7). This specification does not support this functionality.

3.2 Confidentiality, Integrity, Privacy

In this specification, the connection to the SCS can be protected, i.e., HTTPS protocol is used on certain ports (cf. chapter 2.3).

3.3 Local server socket

The SCS server socket uses a server socket that is listening on the localhost interface. The implementers must make sure that connections to the SCS are allowed only from the localhost, i.e., connections from other devices should not be allowed. See also section 3.9 on Malware.

3.4 End user permission

The SCS should request for end user approval each time it receives a request for digital signature generation, i.e., it should not silently sign any content. Therefore the SCS will always show the certificate selection dialog when a signature creation request is received.

3.5 Identifying origin

The SCS should indicate to the end user the origin of the signature creation request. At its simplest terms, it can show the content of the HTTP header Origin, which is sent by the browser when it is making the CORS request to the SCS.

3.6 Non-repudiation signatures

In order to make non-repudiation signatures more meaningful, the end user should be able to approve the content that is being signed. In practice, the SCS should show the content that is being signed to the end user so that the end user can inspect what is being signed. However, for instance binary or xml content cannot be displayed to the end user in understandable form. The current specification allows that content can be signed without showing the end user what is being signed.

3.7 Nonce

In order to prevent the SCS to sign data that is completely given by the requestor, the SCS could be able to add nonce data to the to-be-signed data before it is being signed.

This is typically not possible as it would break that signing procedure. However, it is possible with XML Signatures where the <Signature> element can be appended with optional <Object> elements without breaking the application level logic. However, the current specification does not support this functionality.

3.8 Certificate types

Typically in PKI, the end entity certificates have three major use cases:

- authentication, where the certificate is used to authenticate the user (authentication is typically done by digitally signing a piece of data that has been constructed from nonce values generated by the parties involved)
- decryption, where the key is used to send encrypted data to the holder of the key (encryption is typically done by a shared key that has then been encrypted with recipient's public key)
- non-repudiation, where the certificate used to digitally sign a piece of data (this is analogous to handwritten signature, where the signer enters into an agreement with another party)

The current specification does not limit the use of the certificates based on their intended usage but only digital signatures are supported (authentication, non-repudiation) but the decryption (unwrap) functionality is not supported.

3.9 Malware

As the SCS exposes a server socket on localhost (see section 3.3), any local application, e.g., malware, is able to send signature creation requests to the SCS. As the end user is always prompted with a certificate selection dialog, these requests are not silently processed, but it will be annoying and essentially a denial of service attack as the end user will get these dialogs every time there is a request (unless the SCS is already processing a request). The current specification does not address this threat.

One solution to address this threat is that the SCS has an admin interface (UI), where there would be an authorization view. In the authorization view, there would be an authorization token shown (e.g., 20 alphanumeric values). The first HTML5 application that tries to use the SCS interface in a particular browser, the SCS would send a HTTP response 403 Authorization required. The HTML5 application would then have logic to ask the end user to provide the authorization token. The end user would go to the authorization view, copy the token, and paste it to the dialog the HTML5 application would be showing. The HTML5 application would then give the token to the SCS and the SCS would set a permanent cookie with the browser. Any subsequent request to the SCS by the browser (regardless of the origin of the HTML5 app) would contain this cookie, which SCS would always check. This means that the SCS needs to enable the use of credentials in CORS (e.g., use Access-Control-Allow-Credentials header). After the token has been used once in authorization, it is discarded, and the SCS generates a new one.

3.10 Multi-user enviroments

The current specification does not support cases, where the operating system is a multi-user environment. This is due to the fact that if the system has multiple users logged on, there also should be as many SCS instances running as well. As the SCS should be running in predefined port on the localhost interface, it is not possible to distinguish which SCS instance belongs to which logged on user with the current specification.

3.11 Browser policies

CORS access from a web page is typically limited to the same scheme that was used when the corresponding HTML5 application (i.e., web page) was downloaded. In other words, if a web page was downloaded using https, the subsequent CORS requests must also be done using https. Similarly, if the web page was downloaded using plain http, then CORS requests need to be done with http also.

Some browsers (at the writing of this document, Internet Explorer 10 and 11) may by default restrict the access to the localhost interface using CORS mechanism. In this case, the end user should enable access via browser settings or and a system administrator should enable access by changing browser policies remotely.

4 SCS Profiles

This chapter describes the SCS profiles, i.e., operation parameters for SCS instances. For instance, a SCS instance can be said to "implement" RSA signature profile in which case it must follow profiles specifications listed below.

SCS implementations are required to implement support for parameters that underlined.

4.1 Base profile

Abstract profile.

Server port: **53951** (default http port), **53952** (default https port), 23123 (1st backup http port), 23124 (1st backup https port), 8088 (2nd backup http port), 8089 (2nd back https port). The SCS shall use two of the listed ports, one for http and one for https based access. If one is already in use, it shall attempt to use another port. The HTML5 application shall scan the ports in application specific order, and use the one that is discovered. The SCS may use unspecified ports as well, but the HTML5 application must be aware of these ports.

HTTP methods: <u>GET</u>, <u>POST</u>

HTTP URI: <u>/sign</u>, <u>/version</u>

4.2 RSA signature profile

Extends Base profile.

Hash algorithms: "<u>SHA1</u>", "<u>SHA256</u>" (default), "SHA384", "SHA512"

Signature algorithms: "<u>SHA1withRSA</u>", "<u>SHA256withRSA</u>" (default), "SHA384withRSA", "SHA512withRSA"

Content types (to be signed data): "<u>data</u>", "<u>digest</u>"

Signature types: "<u>signature</u>"

4.3 ECDSA profile

Extends Base profile.

Hash algorithms: "<u>SHA1</u>", "<u>SHA256</u>" (default), "SHA384", "SHA512"

Signature algorithms: "<u>SHA1withECDSA</u>", "<u>SHA256withECDSA</u>" (default), "SHA384withECDSA", "SHA512withECDSA"

Content types (to be signed data format): "<u>data</u>", "<u>digest</u>"

Signature types: "<u>signature</u>"

Annex A: SCS module for javascript

SCS module for javascript can be used to communicate with the SCS. It encapsulates the discovery of the SCS service (the SCS URL), and the use of the sign interface.

- jQuery [http://jquery.com/],
- json2.js [http://www.JSON.org/js.html], and
- a browser that supports CORS [http://enable-cors.org/client.html].

Latest version of the SCS module for javascript can be downloaded from:

- https://developer.fineid.fi/scs/

```
/*******************************************************************
 *   Signature Creation Service module
 *   Version 1.0.1 (2015-05-11)
 *   Copyright 2015 Vaestorekisterikeskus
 *   Pekka Laitinen <pekka.laitinen@vrk.fi>
 *
 *   SCS modules Requires:
 *    - jQuery: https://jquery.com/
 *    - JSON-js: https://github.com/douglascrockford/JSON-js
 *
 *******************************************************************/

var SCS = (function($,JSON) {

   /************************************************************/
   /**** SCS module private methods ***************************/
   /************************************************************/

   // setup initial parameters
   var my = {};
   var secure_urls = [
            "https://localhost.fineid.fi:53952",
            "https://localhost:53952",
            "https://127.0.0.1:53952"
            "https://localhost.fineid.fi:23124",
            "https://localhost:23124",
            "https://127.0.0.1:23124"
            "https://localhost.fineid.fi:8089",
            "https://localhost:8089",
            "https://127.0.0.1:8089"
           ];
   var plain_urls  = [
            "http://localhost.fineid.fi:53951",
            "http://localhost.fineid.fi:23123"
            "http://localhost.fineid.fi:8088"
           ];
   var urls = null;
   var url = null;
   var initialized = false;

   // detect which port is in use
   console.log("SCS: initiating automatic service discovery...");
   autodiscovery(0);

   /**
    * Autodiscovery of the SCS URL on localhost from defined port list.
    */
   function autodiscovery(i) {
      /*
         We need to check whether the document was retrieved
         over http or https as CORS can typically be only be
         done to over https if the document was retrieved over
         https.
```

```
     */
    if (i==0) {
       if (window.location.protocol=="https:") {
          urls = secure_urls;
       } else {
          urls = plain_urls;
       }
    }
    if (i>=urls.length) {
       if (url!=null) {
          console.log("SCS: A service was found at "+url+");
          console.log(“     but we are not sure it is a SCS service.");
       } else {
          console.log("SCS: SCS service not found");
       }
       initialized = true;
       return;
    }

    $.ajax({
       method: "GET",
       url: urls[i]+"/version",
       crossDomain: true,
       dataType: "json",
       settings: { timeout: 100 }
    })
    .done(function(response) {
       url = urls[i]+"/sign";
       initialized = true;
       init(response);
       console.log("SCS: found SCS service at port "+port);
    })
    .fail(function(resp) {
       if (resp instanceof Object) {
          if (resp.status==200) {
             // We are because SCS implementation does not support
             // version check (like early versions of Fujitsu DigiSign.
             // If status is 200 then there was a server responding,
             // but we are not 100% sure that it is SCS service.
             url = urls[i]+"/sign";
             console.log("SCS: "+url);
             console.log("SCS: Found a service in "+url+",
             console.log("     but we are not 100% sure it is SCS service.");
             console.log(“     We use this port if no valid SCS service is found.");
          }
       }
    });
    autodiscovery(i+1);
 }

 /**
  * Initialize the SCS service parameters.
  */
 function init(response) {
    // store possible version parameters of SCS service, i.e., is
    // selector supported, SCS version, supported HTTP methods, etc.
 }

 /**
  * Do SCS signing request based on request parameters.
  */
 function sign_localhost(callback,request) {
    console.log("URL: "+url);
    $.ajax({
       url: url,
       type: "POST",
       crossDomain: true,
       data: JSON.stringify(request),
       contentType: "application/json",
       dataType: "json"
       })
       .done(function(response) {
          doCallback(callback,response);
```

```
         })
         .fail(function() {
            var response = new Object();
            response.ok=false,
            response.reasonCode=500,
            response.reasonText="SCS not responding"
            doCallback(callback,response);
         });
   }

   /**
    * Check if SCS module is initialized (i.e., is finished
    * with autodiscovery of the port).
    */
   function isInitialized() {
      return initialized;
   }

   /**
    * Single point to callback application code.
    */
   function doCallback(callback,response) {
      try {
         response = validate(response);
         callback(response);
      } catch (ex) {
         // This exception should've been catched on caller side!
         console.log(ex);
      }
   }

   function validate(response) {
      // here we can do a sanity check for the response
      return response;
   }

   /*************************************************************/
   /**** SCS module methods ***********************************/
   /*************************************************************/

   /**
    * Check is SCS module is ready.
    */
   my.isAvailable = function() {
      return isAvailable();
   }

   /**
    * Request signature from SCS.
    */
   my.sign = function(callback,
                      content,
                      contentType,
                      hashAlgorithm,
                      signatureType,
                      selector) {

      if (!isInitialized()) {
         console.log("SCS: SCS module not yet initialized");
         var response = new Object();
         response.ok=false,
         response.reasonCode=500,
         response.reasonText="SCS module not yet initialized"
         setTimeout(function() { doCallback(response); },10);
      }

      var request = null;
      if (content instanceof Object) {
         request = content;
      } else {
         request = new Object();
         request.version="1.0";
         request.content = content;
         if (contentType) request.contentType = contentType;
         if (hashAlgorithm) request.hashAlgorithm = hashAlgorithm;
         if (signatureType) request.signatureType = signatureType;
         if (selector) request.selector = selector;
      }

      console.log("SCS: Doing signature with SCS service...");
      sign_localhost(callback,request);
   }

   /**
    * Set SCS port manually (e.g., in the case if port autodiscovery fails).
    */
   my.setURL = function(newurl) {
      url = newurl;
   }

   return my;

}(jQuery,JSON));
```