SWIFTNet

Digital Signatures Using SWIFT Certificates

Implementation Guide for SWIFTNet Link and Alliance Gateway

This guide explains how to use SWIFTNet Link or Alliance Gateway to sign and verify XML DSIG signatures using certificates stored on HSM.

13 March 2020

Table of Contents

Preface

Purpose of this document

This guide explains how to use SWIFTNet Link or Alliance Gateway to sign and verify XML DSIG signatures using certificates stored on HSM.

Intended audience

This document is intended for software developers:

- Who design and implement applications that need to sign XML documents
- Who design and implement communication interfaces and messaging interfaces
- Who design and implement middleware interfaces communicating with communication interfaces and messaging interfaces

This document assumes that the reader is familiar with SWIFTNet.

Related documentation

- *SWIFTNet Link Interface Specifications*
- *SWIFTNet Service Design Guide*
- *ISO 20022 Business Application Header Message Usage Guide*
- *W3C Signature Syntax and Processing (Second Edition) W3C Recommendation 10 June 2008*

Significant changes

This version of the document contains a new section (section 6), called Generating a Digital Signature Using Alliance Gateway Remote API (RA)/Java APIs.

1 Introduction

Background

Currently, SWIFTNet supports digital signatures that are generated (and verified) by calling the SWIFTNet Link (SNL) APIs, and uses certificates stored on a SWIFT Hardware Security Module (HSM)[1]. The signatures have a proprietary format, therefore the SWIFTNet Link calls are used to generate as well as verify the signatures.

The SWIFTNet Link APIs are available either through direct integration with SWIFTNet Link or by means of Alliance Gateway.

In general, the generation of a digital signature is typically composed of the following steps:

1. Identify the data that needs to be signed.
2. Construction of the signature: prepare the XML block that will be provided to the algorithm to generate the signature value. This block may contain a number of parameters such as (a reference to) the signing algorithm, the digest algorithm, and so on. It also contains one or more references to the signed data element(s) and their digests.

   Typically, most parameters are fairly static, except for the digests that are computed on the data to be signed and thus are different for each message. In order to compute the digests, other processing may be necessary, such as canonicalisation of XML payloads before calculating the digest.
3. Prepare the signature request. This includes the specification of the distinguished name (DN) of the certificate that will be used for signing. This step is specific to an implementation. This document describes how to prepare this request to use SWIFTNet Link to generate the signature.
4. Generate the signature. In practice, this means that an API is invoked to which the prepared Signature Request XML block is passed, and the resulting signature is returned with the signature value present. This API is provided by SWIFTNet Link, which interacts with the HSM to find the certificate and perform the signature computation.

The verification process is performed in similar steps.

Today, when sending over SWIFT, a "transport" signature is mandatory and therefore SWIFTNet Link already supports the APIs to generate signatures. The "transport" signature is used as the business signature for services that understand the SWIFTNet signatures. Currently, the SWIFTNet Link API allows using signing and verifying in the following ways:

- While sending/receiving - in this mode, the message (or file) with a signature-before-sign is passed to SWIFTNet Link, which performs all necessary steps to sign and then automatically to send the signed message or file over the SWIFT network. The receiving SWIFTNet Link automatically performs the corresponding verification steps[2].
- Manual mode - in this mode, the application prepares the signature elements and invokes the SWIFTNet Link only to calculate the signature value, and the resulting signed message is passed back to the application. The application then invokes the send API to send the message over the SWIFT network.

This document

This document describes how to use the manual mode of SWIFTNet Link or Alliance Gateway to generate a signature in an open PKI format rather than the proprietary SWIFT format. Such signatures can be used in the payload of a message (or file), such as ISO20022 messages, or in messaging standards such as SOAP. Such use may be also required in a specific business context. An example is the signature that is required in the

[1] Required for the exchange of live traffic.

[2] Typical set-up.

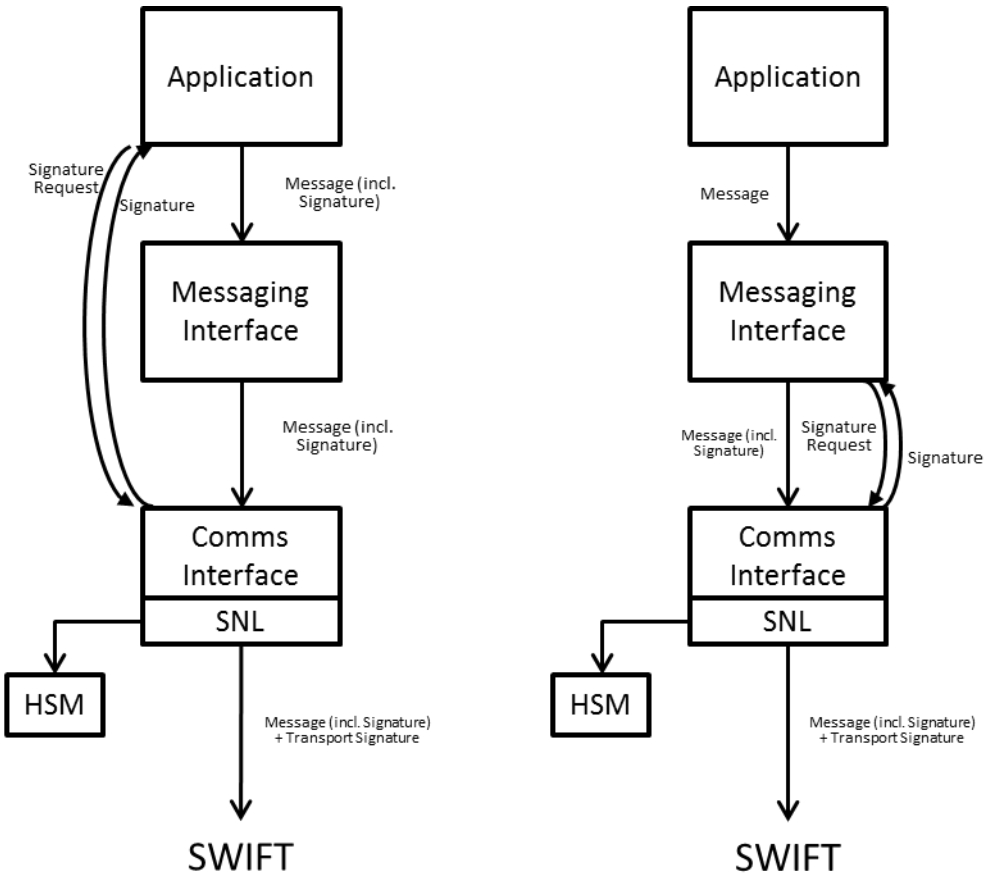
Business Application Header (or the File Business Header, for files) in the context of T2 or T2S.

Note that applications that generate a signature for use in the payload (for example, the Business Application Header with signature, as used for T2 or T2S), can then send the resulting message (or file) over the traditional SWIFT messaging services (InterAct or FileAct) using a messaging interface. In this case, a "transport" signature will also be present. This signature is different from the "business" signature present inside the payload (when used).

As described above, the SWIFTNet Link will typically generate the "transport" signature when sending and the SWIFTNet Link of the receiver will verify the signature when receiving. However, note that the SWIFTNet Link will not automatically generate nor verify the "business" signature present in the payload, because SWIFTNet Link does not know the specific usage by each application or business context.

Example of a use case

The following picture shows two applications that require a Signature made with a private key residing in the HSM connected to SWIFTNet Link.



The application on the left hand side of the diagram uses the communication interface (for instance an Alliance Gateway) to invoke the signature on SWIFTNet Link. Once the Signature Request returns with the Signature, the application inserts the Signature within the payload of the message and then sends the message by means of a Messaging Interface (such as Alliance Access) to SWIFT as usual. A transport signature is added as instructed by the Messaging Interface.

The application on the right-hand side of the diagram sends the message just like any other message to the Messaging Interface. The Messaging Interface then invokes the signing on SWIFTNet Link by means of the communication interface, inserts the resulting Signature within the message, and sends the message to SWIFT. A transport signature is added as instructed by the Messaging Interface.

Various other combinations are possible, such as invoking the Signature Request by means of the Messaging Interface, depending on the features offered by the products used.

This document describes how the application (business application, middleware, or messaging interface) has to construct the Signature Request and how the resulting Signature appears. This document does not describe the further subsequent use of the Signature, for instance, to insert the Signature in the message payload and then send it.

Signing and verifying description

The format of the Signature is described in section 2, Construction of the Signature.

Examples in this document - the Signature within the Business Application Header

This document uses examples of a signature within a Business Application Header (head.001.001.01), for example, as required in the context of T2 or T2S.

All examples are provided for illustration purposes - field contents shown in the examples may or may not reflect valid values for an actual implementation.

Note *In general, the use of the Business Application Header (BAH) does not mandate the use of a business-level signature. It is possible to use the BAH without using the optional signature element, and in that case applications do no need to worry about generating or validating such signatures. In the case of T2 or T2S though, the T2/T2S operator has decided to use the signature field of the BAH, and therefore the sending and receiving applications need to use it.*

2 Construction of the Signature

Initial steps

The application creates the XML document(s) that is to be signed.

The application then creates the Signature as shown in the example. In particular, the application creates the appropriate Reference element(s) within the SignedInfo of the Signature.

Initial steps example

Note *The example is an ISO20022 message with a signature in the Business Application Header. The creation of the Reference elements within the SignedInfo is the responsibility of the application as shown in the example within this document.*

Assume that the application is sending a message reda.010.001.01 that has a Business Application Header.

```
<AppHdr xmlns="urn:iso:std:iso:20022:tech:xsd:head.001.001.01">
  <Fr>
    <FIId>
        <FinInstnId>
        <BICFI>CSDPARTCPNT</BICFI>
        <Othr>
          <Id>CSDBICIDXXX</Id>
        </Othr>
      </FinInstnId>
    </FIId>
  </Fr>
  <To>
    <FIId>
        <FinInstnId>
        <BICFI>SETTLSYST2S</BICFI>
        <Othr>
          <Id>CSDBICIDXXX</Id>
        </Othr>
      </FinInstnId>
    </FIId>
  </To>
  <BizMsgIdr>SENDERSREFERENCE</BizMsgIdr>
  <MsgDefIdr>reda.010.001.01</MsgDefIdr>
  <CreDt>2014-01-01T09:30:47Z</CreDt>
  <Sgntr>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <ds:SignatureMethod
          Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
        <ds:Reference URI="">
          <ds:Transforms>
            <ds:Transform
              Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
              <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
             Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            +OS/MM1NBTiOZWWpvzOWkfRjyP2/F1lg9P+zvC+Gulk=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference>
          <ds:Transforms>
              <ds:Transform
               Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
```

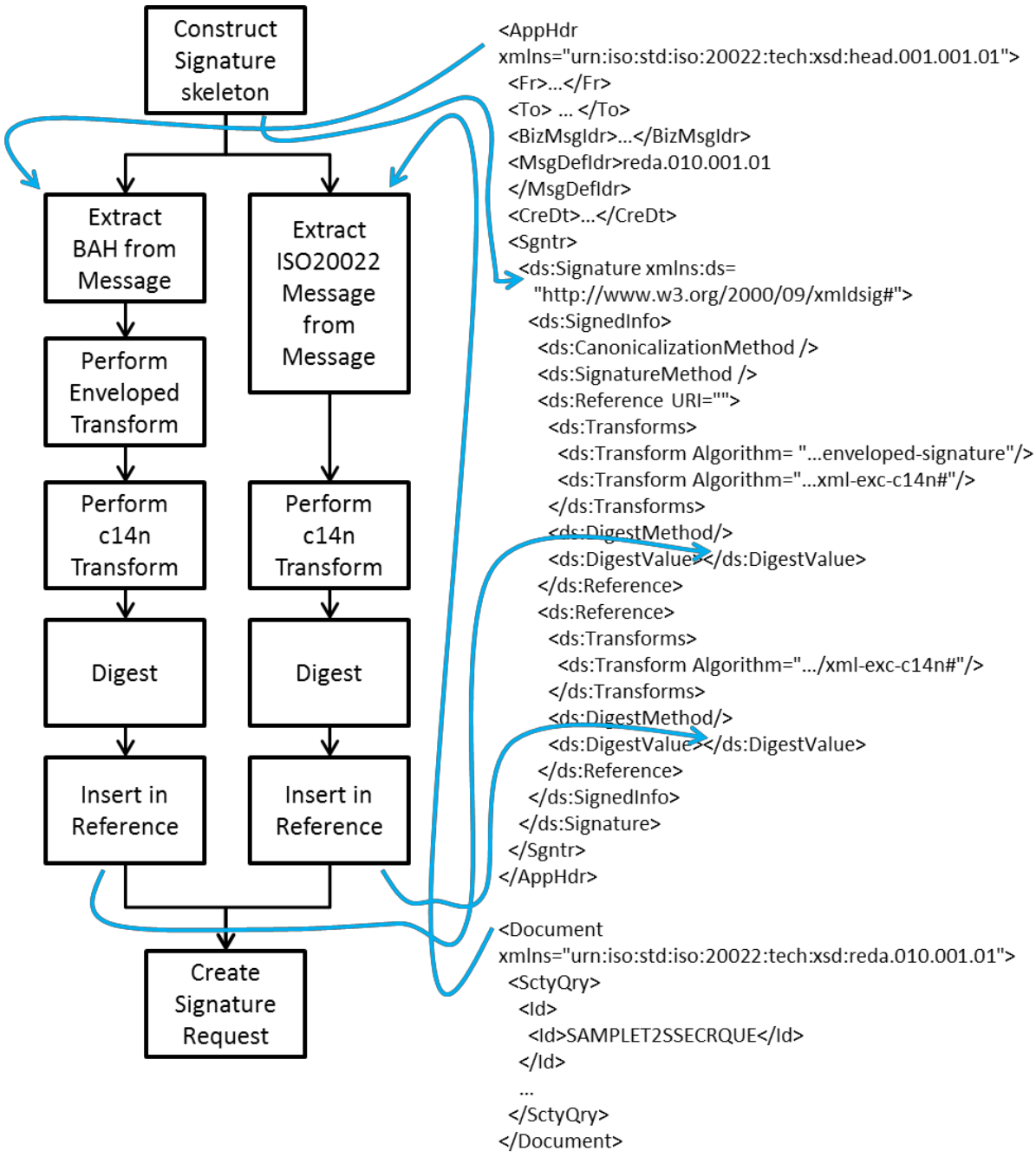
```
            </ds:Transforms>
            <ds:DigestMethod
              Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
            <ds:DigestValue>
              QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
            </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
    </ds:Signature>
    </Sgntr>
</AppHdr>

<Document xmlns="urn:iso:std:iso:20022:tech:xsd:reda.010.001.01">
    <SctyQry>
    <Id>
      <Id>SAMPLET2SSECRQUE</Id>
    </Id>
    <ReqTp>
      <Id>SECR</Id>
      <SchmeNm>Particular</SchmeNm>
      <Issr>ID22</Issr>
    </ReqTp>
    <SchCrit>
      <IsseDt>
        <FrDt>2014-01-01</FrDt>
      </IsseDt>
    </SchCrit>
    </SctyQry>
</Document>
```

The example shows that all elements are fixed, except the content of the two DigestValue elements within the SignedInfo.

The first value, of the Reference with URI="", is calculated on the AppHdr element. As indicated in the Reference, two Transform algorithms are performed on this element. One Transform algorithm to remove the Signature from the digest calculation (the enveloped- signature transform), and the second Transform algorithm is the Exclusive XML Canonicalisation algorithm.

The second value, of the Reference without the URI attribute, is calculated on the Document element. In this case one Transform algorithm, the Exclusive XML Canonicalisation, is executed.

The following diagram shows what the application must perform:



It is left to the application or Messaging interface to integrate the appropriate toolkit offering the XML Exclusive Canonicalisation algorithm. The input for the XML Exclusive Canonicalisation algorithm is the AppHdr element for the first canonicalisation and the Document for the second canonicalisation.

Create Signature Request

Creating the request depends on the interface used by the communication interface or messaging interface.

In the following section, the SWIFTNet Link interface is explained.

3 Signing Using SignEncrypt

SignEncrypt input parameter

Note *The description uses the example from the previous section, the BAH signature. The same processing occurs for other signatures.*

The SignEncrypt API has as input parameter an XML document as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<SwSec:SignEncryptRequest xmlns:SwSec="urn:swift:snl:ns.SwSec">
  <SwSec:AuthorisationContext>
    <SwSec:UserDN>cn=john-smith, o=csdpartc,o=swift</SwSec:UserDN>
  </SwSec:AuthorisationContext>
    <SwSec:SecToSecureData>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <ds:SignatureMethod
          Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
          <ds:Reference URI="">
          <ds:Transforms>
            <ds:Transform
              Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
              <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
             Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            +OS/MM1NBTiOZWWpvzOWkfRjyP2/F1lg9P+zvC+Gulk=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference>
          <ds:Transforms>
              <ds:Transform
                 Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
          </ds:DigestValue>
        </ds:Reference>
        </ds:SignedInfo>
    </ds:Signature>
    </SwSec:SecToSecureData>
</SwSec:SignEncryptRequest>
```

The SecToSecureData contains at least one of the following child elements:

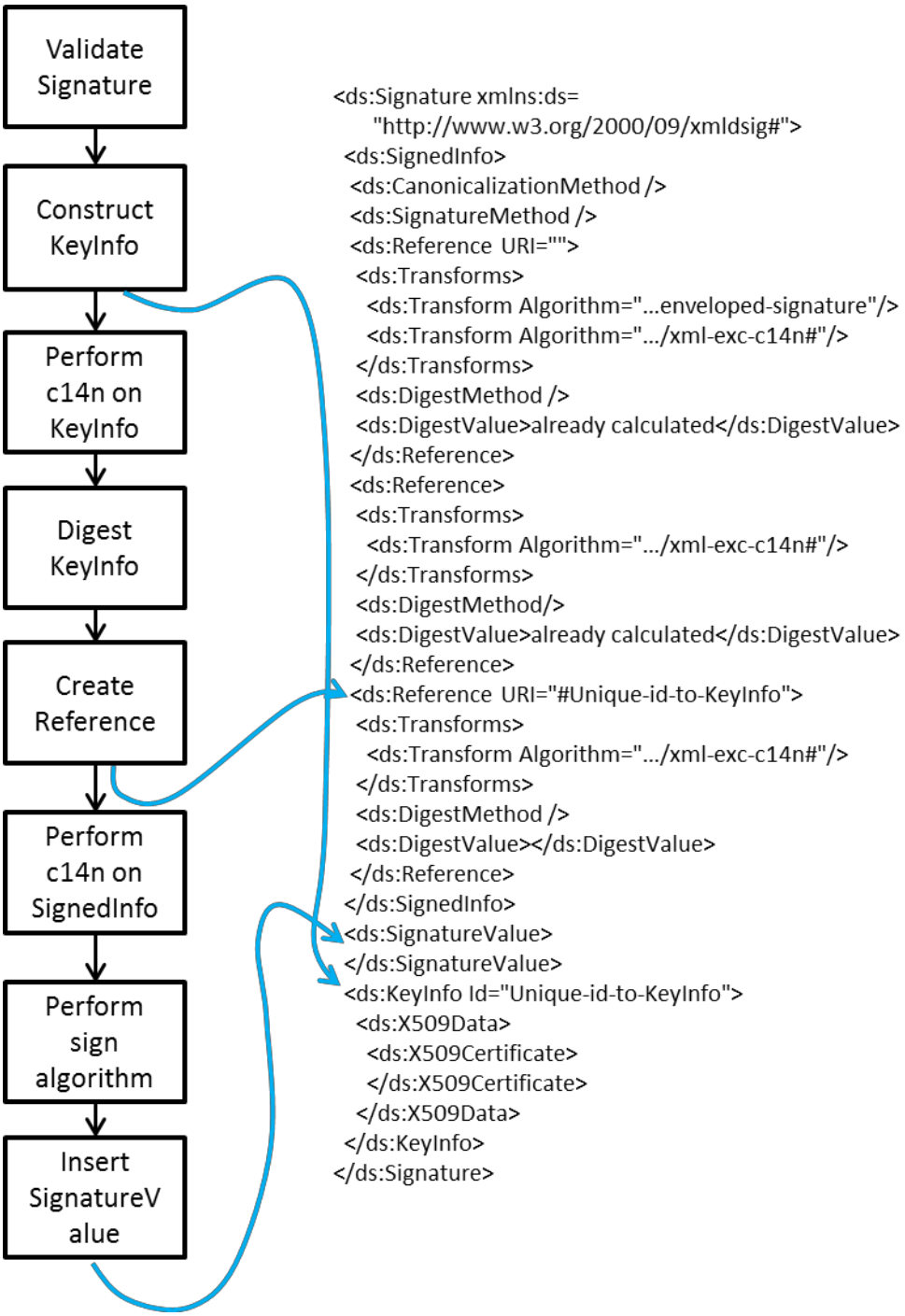
- SwSec:SignatureList
- SwSec:Crypto
- ds:Signature

The SwSec:SignatureList or SwSec:Crypto are SWIFTNet specific formats used for signing InterAct and FileAct traffic. The ds:Signature is a DSIG signature, according to the W3C.

Note *Besides the signature, the SecToSecureData also contains the data to be signed. As shown in the example, in case of a ds:Signature, no further data is required. Indeed, the data to be signed is the SignedInfo.*

SignEncrypt processing

SWIFTNet Link performs the following processing:



The Validate Signature checks that the ds:Signature uses the expected input format:

- Namespace is equal to http://www.w3.org/2000/09/xmldsig#
- KeyInfo is absent.
- SignatureValue is absent.
- CanonicalizationMethod is equal to http://www.w3.org/2001/10/xml-exc-c14n#
- SignatureMethod is equal to http://www.w3.org/2001/04/xmldsig-more#rsa-sha256

Note *The Reference elements found within the ds:Signature are not validated, except for well-formedness.*

If successful, then SWIFTNet Link inserts the certificate used to sign and verify in the KeyInfo element. The signer is the same as the UserDN within the AuthorisationContext.

SWIFTNet Link adds within the SignedInfo a new Reference element pointing to the KeyInfo. This makes the ds:Signature compliant to XAdES-BES.

Finally, SWIFTNet Link signs the SignedInfo using the private key on HSM or on disk as identified by the AuthorisationContext.

The result is returned in the output parameter of the API.

SignEncrypt output parameter

The output parameter contains the ds:Signature with the additional elements Reference to KeyInfo, KeyInfo and SignatureValue.

```
<?xml version="1.0" encoding="UTF-8"?>
<SwSec:SignEncryptResponse xmlns:Sw="urn:swift:snl:ns.Sw"
          xmlns:SwGbl="urn:swift:snl:ns.SwGbl"
          xmlns:SwSec="urn:swift:snl:ns.SwSec"
          xmlns:SwInt="urn:swift:snl:ns.SwInt">
    <SwSec:SecSecuredData>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <ds:SignatureMethod
          Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
          <ds:Reference URI="">
          <ds:Transforms>
            <ds:Transform
              Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
              <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
             Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            +OS/MM1NBTiOZWWpvzOWkfRjyP2/F1lg9P+zvC+Gulk=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference>
          <ds:Transforms>
              <ds:Transform
                 Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference URI="#Unique-id-to-KeyInfo">
          <ds:Transforms>
              <ds:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
          </ds:DigestValue>
        </ds:Reference>
       </ds:SignedInfo>
       <ds:SignatureValue> IKDs7kwX14CxK3AZ1ojJLr35A4eU/98tp/10KFQTtPOwR5WCKyx
      4I05ZV1IljOpEgpkt6xejXhshaEnNBD5B5PII1VN6mviJJjU/njGikNeXzi1Djei2dPEap
      nPX1f26UnQcgYTAqaSVwAnIR7L8/W2UeT8J9z8Rd1OebYV5xE8jVehbgMcAmJwv2rC/c2d
      UkUe2/eBU0APyWGCgKawxGGAPLP3AS4+Mp0ODKlVp08rUzVOF+pFF/1dBknlK/v0dWkDdj
      YvwFRvZhHXue/PYvMNtQBytMUUdB1MiQrNX0jSCE6Y2nljhTXdcrb2lgfgwCclB6xArBRV
      WfMa0kQVQ4Q==
```

```
        </ds:SignatureValue>
        <ds:KeyInfo Id="Unique-id-to-KeyInfo">
          <ds:X509Data>
            <ds:X509Certificate>MIID1DCCArygAwIB   </ds:X509Certificate>
          </ds:X509Data>
        </ds:KeyInfo>
      </ds:Signature>
      </SwSec:SecSecuredData>
 </SwSec:SignEncryptRequest>
```

Processing by the application

The application can take the ds:Signature from the SecSecuredData and put it at the appropriate location in the message. The application may check that the only changes in the input ds:Signature is the addition of a Reference within the SignedInfo pointing to the KeyInfo, the KeyInfo element and the SignatureValue.

4 Verifying Using VerifyDecrypt

VerifyDecrypt input parameter

Note *The description uses the example from the previous section, the BAH signature. The same processing occurs for other signatures.*

The VerifyDecrypt API has as an input parameter an XML document as follows:

```
 <?xml version="1.0" encoding="UTF-8"?>
<SwSec:VerifyDecryptRequest xmlns:SwSec="urn:swift:snl:ns.SwSec">
   <SwSec:AuthorisationContext>
    <SwSec:UserDN>cn=appli1, o=setllbic,o=swift</SwSec:UserDN>
  </SwSec:AuthorisationContext>
  <SwSec:SwiftVerifiedRevocation>FALSE</SwSec:SwiftVerifiedRevocation>
  <SwSec:SecSecuredData>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <ds:SignatureMethod
          Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
         <ds:Reference URI="">
          <ds:Transforms>
            <ds:Transform
              Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
             <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
             Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            +OS/MM1NBTiOZWWpvzOWkfRjyP2/F1lg9P+zvC+Gulk=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference>
          <ds:Transforms>
             <ds:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
          </ds:DigestValue>
        </ds:Reference>
        <ds:Reference URI="#Unique-id-to-KeyInfo">
          <ds:Transforms>
             <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>
            QYYVi9JdlsOxjplrW3vIjT8cWYyzYD4ZnnNJ9SH+dvQ=
          </ds:DigestValue>
        </ds:Reference>
       </ds:SignedInfo>
       <ds:SignatureValue> IKDs7kwX14CxK3AZ1ojJLr35A4eU/98tp/10KFQTtPOwR5WCKyx
      4I05ZV1IljOpEgpkt6xejXhshaEnNBD5B5PII1VN6mviJJjU/njGikNeXzi1Djei2dPEap
      nPX1f26UnQcgYTAqaSVwAnIR7L8/W2UeT8J9z8Rd1OebYV5xE8jVehbgMcAmJwv2rC/c2d
      UkUe2/eBU0APyWGCgKawxGGAPLP3AS4+Mp0ODKlVp08rUzVOF+pFF/1dBknlK/v0dWkDdj
      YvwFRvZhHXue/PYvMNtQBytMUUdB1MiQrNX0jSCE6Y2nljhTXdcrb2lgfgwCclB6xArBRV
      WfMa0kQVQ4Q==
       </ds:SignatureValue>
       <ds:KeyInfo Id="Unique-id-to-KeyInfo">
```

```
            <ds:X509Data>
             <ds:X509Certificate>MIID1DCCArygAwIB   </ds:X509Certificate>
            </ds:X509Data>
          </ds:KeyInfo>
        </ds:Signature>
    </SwSec:SecSecuredData>
</SwSec:VerifyDecryptRequest>
```

Note *The example shows that the SwiftVerifiedRevocation is equal to FALSE. This must be the case, because the verification is done on a Signature that was typically not processed centrally at SWIFT. Putting the SwiftVerifiedRevocation status explicitly to FALSE instructs SWIFTNet Link to check the certificate revocation status as part of the verification of the Signature.*

The SecSecuredData contains at least one of the following child elements:

- SwSec:SignatureList
- SwSec:Crypto
- ds:Signature

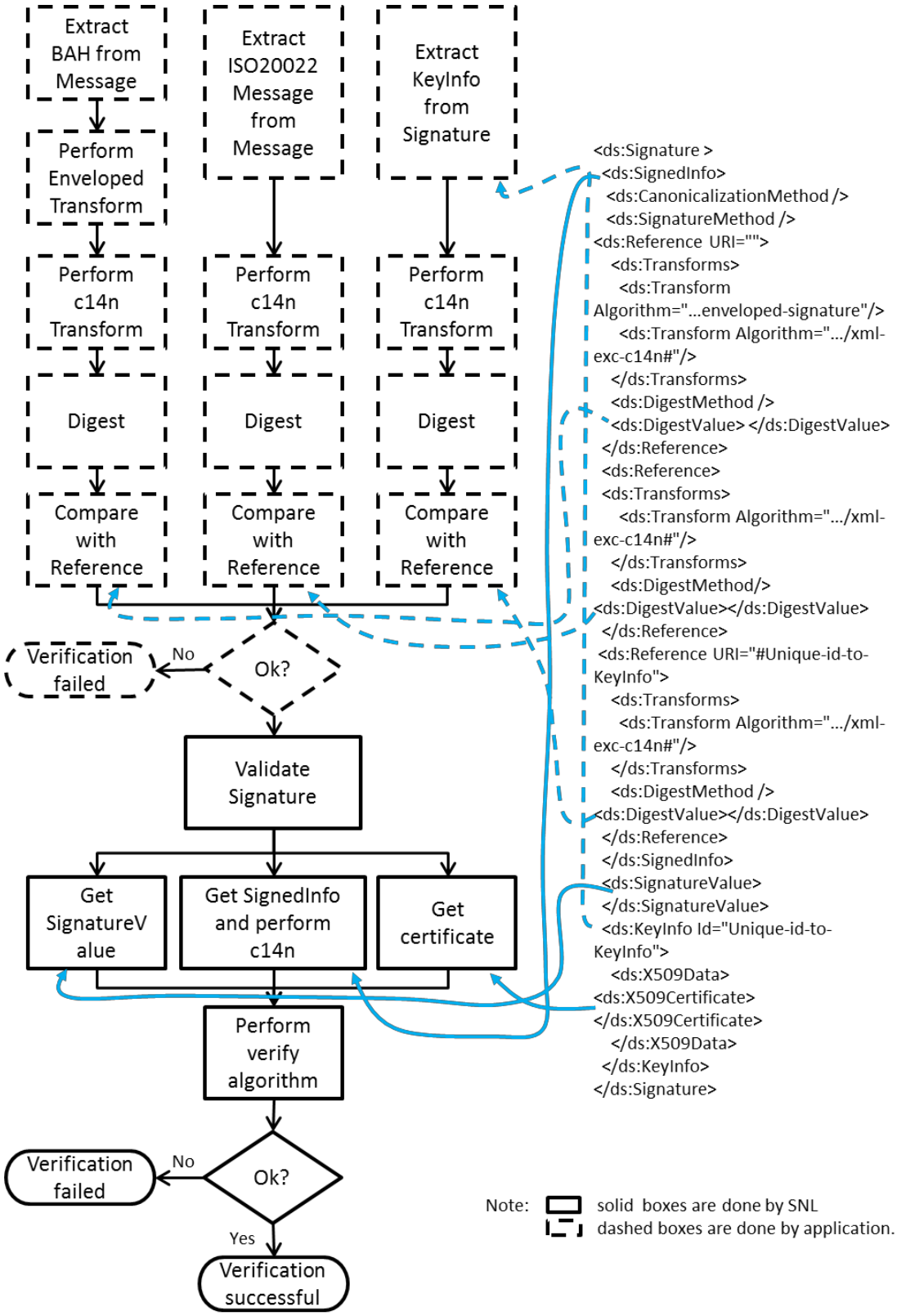
The SwSec:SignatureList or SwSec:Crypto are SWIFTNet-specific formats. The ds:Signature is a DSIG signature, according to the W3C.

Note *Besides the signature, the SecSecuredData also contains the data needed for verification. As shown in the example, in case of a ds:Signature, no further data is required. Indeed, all of the data required for SWIFTNet Link is in the Signature element. It is therefore not necessary to pass the message or file data that is covered by digests within the SignedInfo.*

VerifyDecrypt processing

Verification is similar to signing in the sense that part of the work is done by SWIFTNet Link, and part is done by the application.

The diagram shows the part done by the application using dashed lines. The solid lines are processing steps handled by SWIFTNet Link.



The application validates that the DigestValue within the Reference elements in the SignedInfo matches the data in the message. This is done by locating the data covered by a Reference, performing the Transform algorithms defined in the Reference, and comparing the resulting digest with the one of the appropriate Reference in the SignedInfo. If all Reference elements are successfully checked, the application invokes the VerifyDecrypt API on SWIFTNet Link.

The Validate Signature checks that the ds:Signature uses the expected input format:

- Namespace is equal to http://www.w3.org/2000/09/xmldsig#
- KeyInfo is present and contains a X509Certificate
- SignatureValue is present
- CanonicalizationMethod is equal to http://www.w3.org/2001/10/xml-exc-c14n#
- SignatureMethod is equal to http://www.w3.org/2001/04/xmldsig-more#rsa-sha256

If successful, then SWIFTNet Link checks the SignatureValue to be correct. As part of this check the revocation status of the certificate is validated as well because the SwiftVerifiedRevocation was equal to FALSE (if SwiftVerifiedRevocation is equal to TRUE, the revocation status is not taken into account). The certificate must be a SWIFTNet PKI certificate that is still valid.

If all checks are successful, SWIFTNet Link indicates Verification Successful.

5 SWIFTNet Link Developer Information for Digital Signature

5.1 SwCall API: Client-Side API

Description

SwCall() is called by the client process to perform a data exchange. As for any call, the next instruction after the SwCall is invoked on function return of the SwCall().

The SwResponse is either provided by the local SWIFTNet Link (for example, Sw:InitRequest) or by a remote server.

The actual behaviour of the application, that is, whether the application works in a synchronous mode with the server, is encoded in the input parameter.

As an example, if the input parameter contains an SwInt:ExchangeRequest, then the application is blocked until a response from the server processing the SwInt:ExchangeRequest is returned. If the input parameter contains an SwInt:SendRequest, then the application works asynchronously with the server.

```
SwCallStatus  SwCall (
 SwRequest IN,  XML[SwCallRequest]
 SwResponse OUT,  XML[SwCallResponse]
)
```

Arguments

Argument	Description
SwRequest	Contains the request.
SwResponse	Contains the response and/or the error codes.

5.2 SwSec:SignEncrypt

Description

This primitive signs and/or encrypts the data. It can perform multiple signatures and multiple encryptions on the same set of elements. Encrypted elements are hidden.

	SwCallRequest	SwCallResponse
SwCall	SwSec:SignEncryptRequest	SwSec:SignEncryptResponse
	Contains the authorisation token, the data to be signed and/or encrypted, together with the indication of the crypto operations that should be performed.	Contains the signed and/or encrypted data together with all information required to perform the reverse operation.

SwSec:SignEncryptRequest

Element	Type	Description
SwSec:AuthorisationContext	[1]SwSec:SecurityContext	Authorisation context.
SwSec:SecToSecureData	[1]XML[Any]	Data to be signed and/or encrypted. SecToSecureData contains at least one child, called Signature, SignatureList, or Crypto. Signature, SignatureList, or Crypto child elements are mutually exclusive. If present, the last SignatureList must be of the form "before Sign". Only the last child called SignatureList will be processed.

Element	Type	Description
		The SecToSecureData must also contain the element(s) used to calculate the SWIFTNet Link processable Reference (see SignatureList and Crypto Details). The last Crypto (if present) must be of the form "before Sign/Encrypt". The SecToSecureData must also contain the element(s) to be secured. Only the last child called Crypto will be processed. When requesting signing and encryption, the signature is performed before the encryption takes place. If Signature is present, then all Signature elements will be processed. Each Signature element must be namespace-qualified with the xmlns URI equal to "http://www.w3.org/2000/09/xmldsig#". In each Signature, the KeyInfo and SignatureValue must be absent.

SwSec:SignEncryptResponse

Element	Type	Description
SwSec:SecSecuredData	[0..1]XML[Any]	Data after the signature and encryption. SecSecuredData contains at least one child called "SignatureList" or "Crypto". If present, the Signature elements are modified to contain the KeyInfo with the certificate to be used for verification, a new Reference in the SignedInfo referring to the KeyInfo, and the SignatureValue with the result of the cryptographic operation. If present, the last SignatureList element will be of the form "after Sign" and will contain the cryptographic information to verify the signature(s). If present, the last Crypto will be of the form "after Sign/Encrypt" and will contain the information required to do the reverse operation. Note that the element(s) given in the input parameter SecToSecureData that were encrypted are removed. All elements in the SecToSecureData within the SignEncryptRequest that are not encrypted are returned within the SecSecuredData.
SwGbl:Status	[0..1]SwGbl:Status	Provides error codes in case of failure (SwOperationFailed).

5.3 SwSec:VerifyDecrypt

Description

This primitive has the reverse effect from the sign and encrypt operation.

	SwCallRequest	SwCallResponse
SwCall	SwSec:VerifyDecryptRequest	SwSec:VerifyDecryptResponse
	Contains the authorisation token, the signed and/or encrypted data together with all information required to verify and decrypt.	Contains the decrypted data together with the status of the decrypt and verify.

SwSec:VerifyDecryptRequest

Element	Type	Description
SwSec:AuthorisationContext	[1]SwSec:SecurityContext	Indicates the authorisation context.
SwSec:SwiftVerifiedRevocation	[0..1]Sw:Boolean	By default TRUE. The revocation status of all Signature and Crypto transmitted as children of the Request or Response, and of all Crypto within the FileRequestDescriptor, is checked centrally within SWIFT. The revocation status is not checked centrally within SWIFT when the Signature or Crypto is transported within the Payload. In such cases, this element must be present and set to FALSE, in order to perform the revocation status check locally.

Element	Type	Description
SwSec:SecSecuredData	[1]XML[Any]	Data to be verified and/or decrypted. SecSecuredData contains at least one child called "Signature", "SignatureList", or "Crypto". Signature, SignatureList, or Crypto child elements are mutually exclusive. If more than one SignatureList child is present, only the last one will be processed, except if it has the form "before Sign", in which case no SignatureList if processed.. If only one SignatureList is present, it will not be processed if it is of the form "before Sign". The data required for verification of the signature must be present as well. When there is a failure on a Signature in the SignatureList, then all remaining Signatures will still be processed. If a "Crypto" child is present, it will be of the form "after Sign/Encrypt" and will contain the information required to do the verification and decryption. Only the last child "Crypto" will be processed. If this "Crypto" does not include any encryption, the elements on which the signature applies must be present as well (in the same structure as was used during the Sign/Encrypting). In case of decryption, only the "Crypto" can be present. In case of verification/decryption, the decryption will be done before the verification takes place. If Signature is present, then all Signature elements will be processed. Each Signature element must be namespace qualified with the xmlns URI equal to "http://www.w3.org/2000/09/xmldsig#". The KeyInfo must be a certificate from the SWIFTNet PKI. When there is a failure on a Signature element, then all remaining Signature elements will still be processed.

SwSec:VerifyDecryptResponse

Element	Type	Description
SwSec:SecUnsecuredData	[0..1]XML[Any]	Data after verification/decryption. SecUnsecuredData contains the result of the verify/decrypt operation. If the SecSecuredData contained a SignatureList child, all Signatures within the SignatureList will be verified. All Signatures must be of the form "after Sign". When there was data decrypted, this data is added as a preceding sibling of the first Crypto element within SecUnsecuredData. The last "Crypto" will be in the form "after Verify/Decrypt". This last "Crypto" contains the detail of the operations that were performed. If the SecSecuredData contained a Signature child, the SecUnsecuredData, if present, contains the same content as the SecSecuredData.
SwGbl:Status	[0..1]SwGbl:Status	Provides error codes in case of failure (SwOperationFailed). Note that the Status will indicate the reason identifying each Signature that failed.

5.4 Interface Details Related to C Binding Implementation

Note *SWIFTNet Link 7.4 is available as a native 64-bit application only, unlike previous SWIFTNet Link 7.0.x releases, which were delivered as 32-bit applications. This requires that all applications that interact directly with SWIFTNet Link are upgraded at the same time and compiled in 64 bits.*

The XML interface is used in a C binding environment as follows:

- XML request and response buffers must be null-terminated, and thus restricted to UTF-8 encoding.
- SwCallStatus will have the value (int) 0 for success and (int) 1 for failure. For optional use by applications, the following definitions are made available in a SWIFTNet Link-supplied C header file:

```
typedef int SwCallStatus;
 #define SwOperationFailed
((SwSuccessStatus) 1)
 #define SwOperationSucceed ((SwSuccessStatus) 0)
```

- The function signatures of SwCall and SwCallback are as follows:

```
typedef const char*  SwCallRequest;
typedef
char*  SwCallResponse;

SwCallStatus SwCall (SwCallRequest requestToSend,
SwCallResponse * responseReceived);
SwCallStatus SwCallback (SwCallRequest requestToSend,
SwCallResponse * responseReceived);
```

- The application is responsible for freeing the response buffer on the client side when no longer needed. SWIFTNet Link provides a C binding interface for this purpose. SWIFTNet Link frees the response buffer on the server side. The application must provide a C binding interface that SWIFTNet Link can call to free the server response buffer.

  The function signatures are as follows:

```
void  SwXmlBufferFree( void * ) /* provided by SNL */
void AppXmlBufferFree( void * ) /* provided by application server
*/
```

- This release of SWIFTNet Link supports binary plug-compatibility with other systems emulating SWIFTNet Link calls. SWIFTNet Link client applications must call the following function before initialisation:

```
int SwArguments(int argc, const char **argv)
```

  The argc and argv are the arguments that are passed to the main() function, and contain the arguments from the command line.

- If the application server returns an SwCallStatus value indicating failure (SwOperationFailed), then the content of the response will not be considered and a standard error status will be returned to the requestor.
- If the SwRequest buffer passed to SWIFTNet Link by the application in an SwCall is so badly formed that SWIFTNet Link cannot determine what kind of request it is, then SWIFTNet Link returns an SwResponse of type Status (SwGbl:Status) providing the error code and severity.
- If the response buffer passed to SWIFTNet Link by the application server in an SwCallback is so badly formed that SWIFTNet Link cannot determine what kind of request it is, then SWIFTNet Link returns a status error to the Requestor, and logs the error locally at the server host.

- XML comments located inside the payload (request or response) will be transmitted end-to-end. However, XML comments inserted in other parts of the request or response will not be transmitted over the network.
- Element attributes and types included directly in an SwRequest or SwHandleResponse XML string, unless within the request or response payload, are not transmitted across the network. Any element attributes associated with the payload or within the payload are transmitted as is (as are all data within the payload).
- SwGbl:DateTime is defined as a string containing the datetime in ISO 8601 format. ISO 8601 defines several format options for the date time. SWIFTNet Link returns the date and time in the following ISO 8601 format: YYYY-MM-DDTHH:MM:SSZ (the T is a fixed symbol indicating start of time portion of the string).

Note *A sample SWIFTNet Link client C source code is provided with the SWIFTNet Link run-time software. It can be found in the SWIFTNet Link sample sub-directory (for example, C:\Swift\SNL\Swiftnet\samples\sources). The way to build the application depends on the operating system. Compiler versions are listed in the OS Levels and Patches Baseline document.*

6 Generating a Digital Signature Using Alliance Gateway Remote API (RA)/Java APIs

Introduction

In terms of digital signatures, Alliance Gateway provides a means to remotely use an instance of SWIFTNet Link. The primitives `SignEncrypt` and `VerifyDecrypt` remain identical. Sending such primitives uses the APIs of Alliance Gateway instead of those of SWIFTNet Link.

The APIs of Alliance Gateway Remote API (RA) are available in C++ and in Java. This section explains how to use the Java APIs to send the primitives.

Sample and JAR files

Associated sample code and JAR files are available from SWIFT on request.

6.1 Connectivity

6.1.1 Configuration

The term “Remote APIs” implies a connection. A connection implies authenticating the application calling the APIs. Authentication implies provisioning credentials and authorisations as configuration data in Alliance Gateway.

Term	Description
Identity	The application mandates the provisioning of a message partner profile in Alliance Gateway. The identifier is the message partner name.
Local authentication key	SWIFT mandates using a local authentication protocol each time an application uses an instance of SWIFTNet Link remotely. The protocol is based on message authentication codes (MAC), using a symmetric key and the HMAC/SHA256 signature algorithm. SWIFT also mandates that the application itself computes and verifies the MAC of each exchanged message. Alliance Gateway must comply in the same way as any other connectivity software. Both the application and the message partner profile in Alliance Gateway share a local authentication key. Each message exchanged between the application and Alliance Gateway is signed using the key. The signature is a local message authentication code (LMAC).
Authorisations	A successful authentication grants access to everything authorised by the message partner profile. In the case of digital signatures, it is necessary to provision two types of authorisations: • The authorisation to use the “Relaxed SWIFTNet Link protocol”, also known as “relaxed mode” or “relaxed SWIFTNet Link message format”. • The authorisation to use an explicit list of signature DNs with the relaxed SWIFTNet Link protocol. Each DN of this list corresponds to a PKI profile in the HSM connected to the instance of SWIFTNet Link. Alliance Gateway mandates the declaration of PKI profiles as “real SWIFTNet users”, also known as “SWIFTNet certificates”. Each “real SWIFTNet user” must be individually authorised to be used with the relaxed SWIFTNet Link protocol.

6.1.2 Connection

The APIs define a class `SagHandle` that represents an actual connection. The handle requires the following connectivity data: the hostname of Alliance Gateway, the port number of Alliance Gateway (48002 by default), the file name of the TLS certificate, and the DN of the TLS certificate.

The method `connect` uses the connectivity data to connect to Alliance Gateway. The method `disconnect` cuts the connection and frees any input/output resources.

Example

```
CertificateFactory certificateFactory = CertificateFactory.getInstance ( "X.509" ) ;
InputStream        inputStream        = new FileInputStream ( "File name of the TLS certificate" ) ;
X509Certificate    tlsCertificate     = (X509Certificate) certificateFactory.generateCertificate (
    inputStream ) ;

HandleParameters  handleParam    = new HandleParameters (
  "Host name of Alliance Gateway" ,
  48002                          , // Port number of Alliance Gateway
  tlsCertificate                 ,
  "DN of the TLS certificate"    ) ;
SagHandle  sagHandle ;

try {
  sagHandle = new SagHandle ( handleParam ) ;
  sagHandle.connect() ;
  /* Application is running */
}
catch ( RAException e ) {
  /* The connectivity of Alliance Gateway has rejected the connection */
}
finally {
  if ( sagHandle != null ) {
    sagHandle.disconnect() ; // Riddance of every pending input/output resources
  }
}
```

6.1.3 Client Transaction

A client application performs a request/response transaction by calling the method `call`. This method requires a message to send as a request, and returns the corresponding response as a message.

A message of Alliance Gateway consists of two items: an envelope and a letter. In the scope of the relaxed SWIFTNet Link protocol, the letter contains the primitive exchanged with the instance of SWIFTNet Link.

The envelope contains the metadata exchanged between the application and Alliance Gateway itself. For the digital signatures, the envelope contains the following items:

- The message partner name.
- The local message authentication code (LMAC) computed using the local authentication key of the indicated message partner.
- The identifier of the relaxed SWIFTNet Link protocol.

The class `Message` contains a message. It supports a setter method and a getter method for the letter. It also supports the method `getEnvelope` giving access to the envelope object. The envelope object supports setter methods and getter methods for each field. Those relevant for the digital signatures are:

Method	Purpose
`setApplicationId` `getApplicationId`	Sets and gets the message partner name.

setLocalAuth getLocalAuth	Sets and gets the LMAC.
setMsgFormat getMsgFormat	Sets and gets the identifier of the relaxed SWIFTNet Link protocol: Sag:RelaxedSNL
setContextId getContextId	Options of the relaxed SWIFTNet Link protocol. For the digital signatures, the options are the following keywords, each separated by a white-space character: Advanced Namespace NewErrorCodes

The APIs of Alliance Gateway provide a utility class LMAC to compute and verify LMACs. They support two methods: compute and verify. Both methods have three parameters: the local authentication key, the name of the algorithm that is always HMAC_SHA256, and the message object.

Method	Purpose
compute	Takes the letter and the envelope (except the LMAC) of the message to compute, and sets the LMAC in the envelope.
verify	Takes the letter and the envelope (except the LMAC) of the message to compute an LMAC, and compares this LMAC with the LMAC of the envelope.

Example

```
/* LAU : Local Auhtentication */

String  g_sMessagePartnerNameOfTheApplication =  "Message partner name"            ;
String  g_sLAUKeyOfTheApplication             =  "String of the local authentication key" ;
byte[]  g_byteLAUKeyOfTheApplication          =  g_sLAUKeyOfTheApplication.getBytes (
    StandardCharsets.UTF_8 ) ;

LMAC    g_lmac = new LMAC() ;

String  f_sPrimitiveResponseOfClientTransaction ( String  a_sPrimitiveRequest )
{
  try {

    Message requestMessage     = new Message () ;
    Message responseMessage    = null ;
    String  sPrimitiveResponse ;

    requestMessage.setLetter ( a_sPrimitiveRequest ) ;

    requestMessage.getEnvelope().setContextId     ( "Advanced Namespace NewErrorCodes"    ) ;
    requestMessage.getEnvelope().setMsgFormat     ( "Sag:RelaxedSNL"                      ) ;
    requestMessage.getEnvelope().setApplicationId ( g_sMessagePartnerNameOfTheApplication ) ;

    g_lmac.compute ( g_byteLAUKeyOfTheApplication ,
                     "HMAC_SHA256"  ,
                     requestMessage ) ;

    responseMessage = sagHandle.call ( requestMessage ) ;

    g_lmac.verify ( g_byteLAUKeyOfTheApplication ,
                    "HMAC_SHA256"   ,
                    responseMessage ) ;

    if ( !responseMessage.getEnvelope().getApplicationStatus().equals ( "SUCCESS" ) ) {
      /* SwiftNet Link SwCall API has rejected the primitive with a status. The letter contains the
    status */
    }

    sPrimitiveResponse = responseMessage.getLetter() ;

    return sPrimitiveResponse ;
  }
```

```
  catch ( RAException  e ) {
    /* Alliance Gateway or the connectivity to Alliance Gateway has rejected the message */
  }
}
```

Legal Notices