Mobile XML Signature Component 1.0 Specification

1. Design

This component will provide the ability to digitally sign and verify XML documents. The WSE 3.0 does not currently provide support for signing and encrypting SOAP messages on the .Net compact framework. This component is part of a set of components that will provide the minimum functionality to help fill this gap.

1.1 General approach

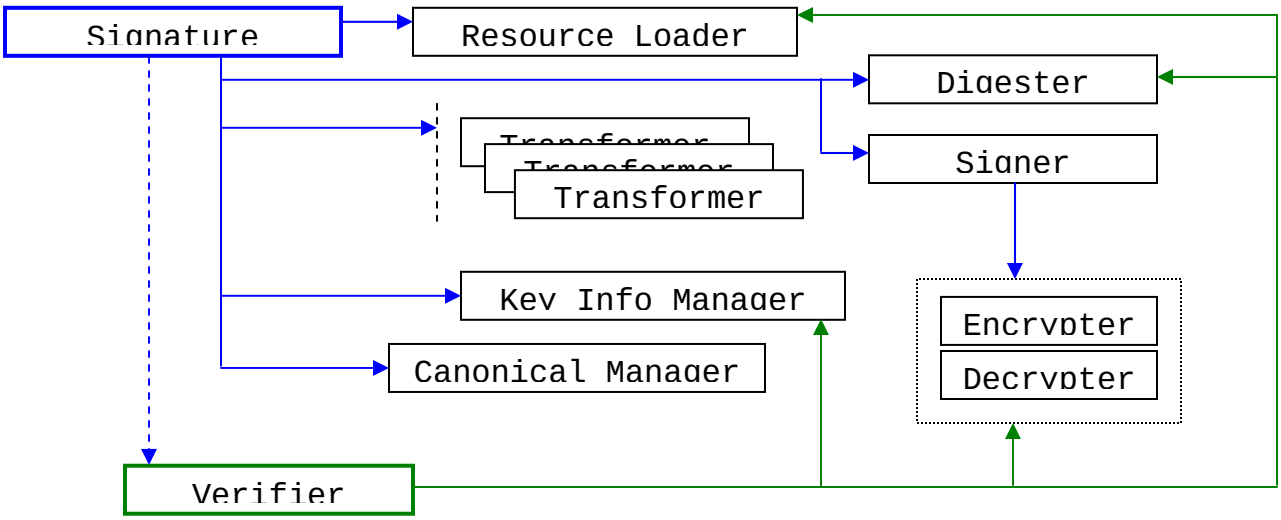
The key to the approach of signing a document is the complete reproducibility of the processes and steps that took part in the overall process of signing. In other words this is a highly defined and deterministic process.

1.1.1 What we need to capture

We need to capture the following capabilities:

1. We need to be able to get the input of which resources are to be signed
   a. This is in the form of a Uniform Resource Identifier (URI)
2. For each resource we calculate the digest through a configurable digest method
   a. We will need to be able to uniquely identify the method that we used to sign the resource.
3. We must have the capability to take all the resources for which we calculated the digests and to bundle them together into a single element to be signed.
   a. Here we would need to ensure that the bundled data could be canonicalized (i.e. put into a canonical form) so that we ensure data determinism.
   b. We then calculate the digest for the whole bundle and we encrypt it.
4. We must have the ability to include keying information (such as the X.509 certificate for the sender, which would include the public key needed for signature verification – currently we only support X.509 certificate)

Here is how the proposed architecture relates to the steps provided above:



The basic flow is as follows:

Signature Manager Flow:

1. A signer will be given a list of resources. For each resource we will also have a list of possible transformers to apply to the resource.
2. For each resource in the list we will load this resource using the configured Resource Loader. This will loaded as either a stream or a byte array
3. Once we have the bytes of the resource we pair it with the specific Digester, which produces the digest for these bytes.
4. Once all the resources have been loaded and signed they will be collected (i.e. their bytes collectively) and then passed through a Cononicalization process, which will bring the whole stream to a canonical format.
5. Once the Cononicalization is done we will take the resulting bytes and create a digest for the whole collection of resources as a single byte stream. This is the signed data stream.
6. This hash code is then signed (hashed and encrypted using a specific encryption algorithm for example) by a specific Signer to produce the final signature of the data.
7. Finally the keying information is included though the Key Info Manager (in our case currently the only option is the X.509 certificate)
8. The outcome of these steps will be an XML element, which can then be included in any document that it is relevant to.

Verification Manager Flow:

1. A verifier will be given an XML element (node) and will read the singed data stream.
2. It will then use the specified digest algorithm (which would be written by the signer into the xml) to recalculate the digest of this signed data stream.
3. Next we calculate the digests of the resources (based on their references) as well as the complete data stream. This will produce the hash code for this stream.
4. We then decrypt the hash code using the X.509 certificate public key.

1.1.2 Configuration Considerations

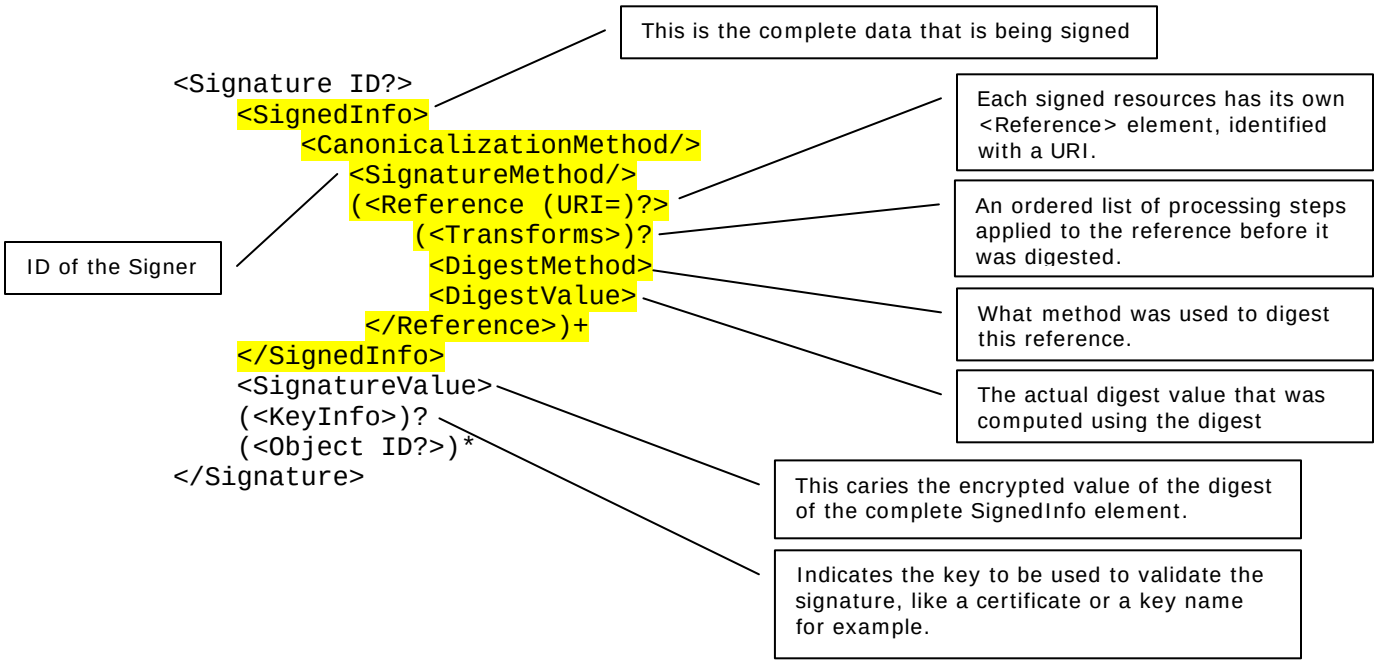
Pieces of the component such as Digester or Encrypter will configurable and (very important) will use as their configuration keys, which will be very specific in some instances as presented by w3.
Take as an example the <SignatureMethod element, which has an attribute Algorithm. This attribute will be set to a very specific widely recognized value such as "http://www.w3.org/2000/09/xmldsig#sha1" but which can be any value as long as both sides recognize it.

1.1.3 Signature considerations

A fundamental feature of XML Signature is the ability to sign only specific portions of the XML tree rather than the complete document. This is relevant when a single XML document may have a long history in which different parties author the different components at different times, each signing only those elements relevant to itself.
This flexibility will also be critical in situations where it is important to ensure the integrity of certain portions of an XML document, while leaving open the possibility for other portions of

the document to change. Consider, for example, a signed XML form delivered to a user for completion. If the signature were over the full XML form, any change by the user to the default form values would invalidate the original signature.

1.1.4 Components of an XML Signature



The whole point of this component is to build the above document element (i.e. <Signature> element) based on input of references and a registry of different methods.

1.1.5 Method registry

One important aspect of this process is the flexibility of the different methods that can be used at all the stages of the signing process. We can have different implementations of transformers, or Digesters, or Encrypters, etc… The idea is that when we are either reading the xml to verify it or when we are producing the signature, we need to be able to properly and deterministically use very specific classes for all these processes. For this purpose we will have a registry (configurable), which will store all such methods that we know of and can use. Each such method/process would be registered under a specific key.
For example, we could be reading the following snippet of xml:

```
<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

But then how would we know what specific implementation to use? We would simply use the "http://www.w3.org/2000/09/xmldsig#sha1" as a key to our registry so that we could lookup the specific Digester as follows:

```
Digester myDigester  =
         registry.GetDigesterInstance("http://www.w3.org/2000/09/xmldsig#sha1");
```

Same idea would apply to the other implementations needed in the complete signing ir verifying process.

1.2 Design Patterns

The **Strategy** pattern is used with the `IThesaurusPersistence` implementation classes. The current implementation of `IThesaurusPersistence`, which is `XmlThesaurusPersistence` plug into the `ThesaurusManager` through the available constructor and is thereafter used by the manager to fetch data.

1.3 Industry Standards

- NET Framework 2.0
- Mobile 5.0

1.4 Required Algorithms

There are no major algorithms involved here but I will address the specifics that were required in the RS. I will also provide a very general overview of the signing and verification process:

1.4.1 Support for SHA1 algorithm

The component will support the SHA 1 algorithm for calculating the digest. .NET already supports SHA 1 through `SHA1CryptoServiceProvider` class which has full mobile support for all the methods. Thus this is what we would do to calculate the hash:

```
byte[] data = new byte[DATA_SIZE];
ubyte[] result;
// This is one implementation of the abstract class SHA1
SHA1 sha = new SHA1CryptoServiceProvider();
result = sha.ComputeHash(data);
```

1.4.2 Support for RSA algorithm

.NET already supports RSA through `RSACryptoServiceProvider` class which has full mobile support for all the methods. Thus this is what we would do to calculate the hash:
Thus this is what we would do to encrypt/decrypt:

```
//Create a new instance of RSACryptoServiceProvider.
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
//Import the RSA Key information. This only needs
//to include the public key information.
RSA.ImportParameters(RSAKeyInfo);
//Encrypt the passed byte array and specify OAEP padding.
//OAEP padding is only available on Microsoft Windows XP or
//later.
return RSA.Encrypt(DataToEncrypt, DoOAEPPadding);
```

Encryption

```
//Create a new instance of RSACryptoServiceProvider.
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
//Import the RSA Key information. This needs
//to include the private key information.
RSA.ImportParameters(RSAKeyInfo);
//Decrypt the passed byte array and specify OAEP padding.
//OAEP padding is only available on Microsoft Windows XP or
//later.
return RSA.Decrypt(DataToDecrypt, DoOAEPPadding);
```

Decryption

1.4.3 Support for DSA algorithm

.NET already supports DSA through `DSACryptoServiceProvider` class which has full mobile support for all the methods. Thus this is what we would do to sign/verify:

```
//Create a new instance of DSACryptoServiceProvider.
DSACryptoServiceProvider DSA = new DSACryptoServiceProvider();
//Import the key information.
DSA.ImportParameters(DSAKeyInfo);
//Create an DSASignatureFormatter object and pass it the
//DSACryptoServiceProvider to transfer the private key.
DSASignatureFormatter DSAFormatter = new DSASignatureFormatter(DSA);
//Set the hash algorithm to the passed value.
DSAFormatter.SetHashAlgorithm(HashAlg);
//Create a signature for HashValue and return it.
return DSAFormatter.CreateSignature(HashToSign);
```

Sign

```
Create a new instance of DSACryptoServiceProvider.
DSACryptoServiceProvider DSA = new DSACryptoServiceProvider();
//Import the key information. DSA.ImportParameters(DSAKeyInfo);
//Create an DSASignatureDeformatter object and pass it the
//DSACryptoServiceProvider to transfer the private key.
DSASignatureDeformatter DSADeformatter = new DSASignatureDeformatter(DSA);
//Set the hash algorithm to the passed value.
DSADeformatter.SetHashAlgorithm(HashAlg);
//Verify signature and return the result.
return DSADeformatter.VerifySignature(HashValue, SignedHashValue);
```

Verify

1.4.4 Creating a signature

Here we will outline the main steps in creating a signature:

1. Load all resource bytes
2. For each resource calculate the digest for each resource
3. Get all of the resource ids and their digests and create a single byte stream
4. Canonicalize the resulting bytes stream
5. Sign the canonicalized resulting byte stream

1.4.4.1 Determine which resources are to be signed.

This will take the form of identifying the resources through a Uniform Resource Identifier (URI).

1.4.4.2 Calculate the digest of each resource.

In XML signatures, each referenced resource is specified through a <Reference> element and its digest (calculated on the identified resource and not the <Reference> element itself) is placed in a <DigestValue> child element like

```
<Reference URI="http://www.abccompany.com/news/2000/03_27_00.htm">
   <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
   <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
</Reference>
<Reference
          URI="http://www.w3.org/TR/2000/WD-xmldsig-core-20000228/signature-example.xml">
   <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
   <DigestValue>UrXLDLBIta6skoV5/A8Q38GEw44=</DigestValue>
</Reference>
```

The <DigestMethod> element identifies the algorithm used to calculate the digest, which in our case would come from the registry.

1.4.4.3 Collect the Reference elements

Collect the <Reference> elements (with their associated digests) within a <SignedInfo> element like:

```
<SignedInfo Id="foobar">
   <CanonicalizationMethod
                    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
   <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
   <Reference URI="http://www.abccompany.com/news/2000/03_27_00.htm">
       <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
       <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
   </Reference>
   <Reference
          URI="http://www.w3.org/TR/2000/WD-xmldsig-core-20000228/signature-example.xml">
       <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue>UrXLDLBIta6skoV5/A8Q38GEw44=</DigestValue>
   </Reference>
</SignedInfo>
```

The <CanonicalizationMethod> element indicates the algorithm was used to canonize the <SignedInfo> element. Different data streams with the same XML information set may have different textual representations, e.g. differing as to white space. To help prevent inaccurate verification results, XML information sets must first be canonized before extracting their bit representation for signature processing. The <SignatureMethod> element identifies the algorithm used to produce the signature value.

1.4.4.4 Signing

Calculate the digest of the <SignedInfo> element, sign that digest and put the signature value in a <SignatureValue> element.

```
<SignatureValue>MC0E LE=</SignatureValue>
```

1.4.4.5 Add key information

If keying information is to be included, place it in a <KeyInfo> element. Here the keying information contains the X.509 certificate for the sender, which would include the public key needed for signature verification. Here is an example:

```
<KeyInfo>
    <X509Data>
        <X509SubjectName>CN=Ed Simon,O=XMLSec Inc.,ST=OTTAWA,C=CA</X509SubjectName>
        <X509Certificate>MIID5jCCA0+gA...lVN</X509Certificate>
    </X509Data>
</KeyInfo>
```

1.4.4.6 An example of a resulting document node

```
<xml version="1.0" encoding="UTF-8"?>
   <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo Id="foobar">
      <CanonicalizationMethod
                   Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      <SignatureMethod
                   Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
      <Reference URI="http://www.abccompany.com/news/2000/03_27_00.htm">
         <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
         <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
      </Reference>
      <Reference
          URI="http://www.w3.org/TR/2000/WD-xmldsig-core-20000228/signature-example.xml">
```

```
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>UrXLDLBIta6skoV5/A8Q38GEw44=</DigestValue>
         </Reference>
      </SignedInfo>
      <SignatureValue>MC0E~LE=</SignatureValue>
      <KeyInfo>
         <X509Data>
            <X509SubjectName>CN=Ed Simon,O=XMLSec Inc.,ST=OTTAWA,C=CA</X509SubjectName>
            <X509Certificate>MIID5jCCA0+gA...lVN</X509Certificate>
         </X509Data>
      </KeyInfo>
   </Signature>
```

1.4.5 Verifying a signature

This is done in two stages:

1.4.5.1 Verify the signature of the <SignedInfo> element.

To do so, recalculate the digest of the `<SignedInfo>` element (using the digest algorithm specified in the `<SignatureMethod>` element) and use the public verification key to verify that the value of the `<SignatureValue>` element is correct for the digest of the `<SignedInfo>` element.
If this step passes we then go deeper into the references themselves

1.4.5.2 Recalculate the digests of references

Here recalculate the digests of the references contained within the `<SignedInfo>` element and compare them to the digest values expressed in each `<Reference>` element's corresponding `<DigestValue>` element.
This is basically a copy of the steps provided for signing the document.

1.4.6 Canonicalizing an xml document

Please note that the following overview has been taken as an excerpt from here:
http://www-128.ibm.com/developerworks/xml/library/x-c14n/
Please us the full document as a reference for more intricate detail of how this should be implemented:

The following steps provide an overview of the whole process:

1. The document is encoded in UTF-8.
2. Line breaks are normalized to "#xA" on input, before parsing.
3. Attribute values are normalized, as if by a validating processor.
4. Default attributes are added to each element, as if by a validating processor.
5. CDATA sections are replaced with their literal character content.
6. Character and parsed entity references are replaced with the literal characters (excepting special characters).
7. Special characters in attribute values and character content are replaced by character references (as usual for well-formed XML).
8. The XML declaration and DTD are removed. (Note: I always recommend using an XML declaration in general, but I appreciate the reasoning behind omitting it in canonical XML form.)
9. Empty elements are converted to start-end tag pairs.
10. Whitespace outside of the document element and within start and end tags is normalized.
11. All whitespace in character content is retained (excluding characters removed during line feed normalization).
12. Attribute value delimiters are set to quotation marks (double quotes).

13. Superfluous namespace declarations are removed from each element.
14. Lexicographic order is imposed on the namespace declarations and attributes of each element.

1.5 Component Class Overview

1.5.1 TopCoder.Security.Crytography.Mobile

SignatureManager

This is the main manager of this component. It is responsible for creating a Digital (xml based) signature as well as being able to verify digital signatures. For signature creation it coordinates the work of reference loaders to load references, transformers to transform each loaded reference, digesters to provide digests for each loaded reference, canonicalizers to bring the data to a standardized canonical form, digester to provide a digest of the canonical form, signer which will sign the resulting canonical form.

This manager is 'smart' enough to know the exact format of the xml that will form the signed document. In contrast, none of the processors know anything about xml. The result of this process will be an xml-based signed element, which will be a digital signature of the data provided by the references.

It will also be responsible for verification of a digital signature (an inverse process to a great extent of signing) so that when given an input consisting of an XML node (or an xml string) it can verify the signature.

This component uses the ProcessRegistry to fetch all the necessary processing elements (like a digester or a transformer) for added flexibility.

This implementation is thread safe as it acts mostly as a utility and all state is invoked in a thread-safe manner. In our case all used subcomponents are required to be thread-safe which means that the manager doesn't have to do any special extras to ensure thread-safety.

NOTE: both the digest output as well as signature must be base64 encoded. We will use .NETs `Convert.ToBase64String()` to convert unsigned bytes to base64 and the reverse will be accomplished with `Convert.FromBase64String()`

ProcessRegistry

This is a registry of processors that can process different aspects of the signing or verifying main process. This is really nothing more than a configurable dictionary of keys, which acts as a factory in creating/fetching the required elements. For example when the SignatureManager needs to create a digest it will ask the registry to fetch it a specific (key based) Digester from the registry.

The registry is initialized through a configuration file and loads up class names, which are, used to (through reflection) instantiate the specific instances. Note that we will use the property-name as the key into our registry and the value will be the instance created from the value in configuration, which is a class name.

Since each class will be instantiated through a default constructor, it will also be the responsibility of the registry to initialize the instantiated object at the time of caller request for the instance, with property values passed in via the IDictionary params parameter which we simply use through reflection by using ach key in the dictionary as the name for the setter that we are looking for.

Implementation of this class is thread-safe since each object is created on demand and thus this is a stack-based creation. In addition to that the actual map using in the registry is immutable after being created so it is a read only based state.

InstantiationVO
This is a simple Value Object class that is used to store the instantiation key for a class (used in the ProcessRegistry) and the corresponding parameters map used to initialize the resulting instance object.
The basic use of this class will be store information about instantiation for some class instance. Thus the "key" variable will be used as an identifier of which class we want to get an instance of, and "params" will hold the parameters (or properties) that should be initialized with the provided values.

IReference
This is a contract for a value object, which contains all the necessary information about how a reference is structured (or to be structured) within a signature. The structure is quite simple: A reference has a URI (required) associated with it as well as a set of transformers (optional), which are ordered, to be used in transforming this reference. It also has an associated digester (required). This will then be passed to the SignatureManager (as a list of references) Note that the reference is not required to actually hold the specific instances of the objects that are associated with it (such as Transformers or Digesters) but rather registry keys.
Implementations must be thread-safe. This is quite simple to accomplish since only getters are exposed in the interface. All data must be set in the constructor.

Reference
This is a simple implementation of the IReference interface, which holds information about a reference.
Implementation is thread-safe as it is immutable.

IDigester
This is a contract for creating a digest based on input byte data. All digester implementations must be thread-safe.

ITransformer
This is a contract for creating a transformation, which will accept input data (either binary or text) and which will then transform this data in some predictable way. All transformer implementations must be thread-safe.

ISigner
This is a more complex processor, which will on occasions take a number of steps to accomplish its task. This is a contract for a signer, which will take input data of bytes and will sign this data (usually a digest followed by encryption followed by base64 encoding to fit it into an xml string)
Usually a signer needs a private key to be able to sign the content of the data, but this contract doesn't limit the application of generating a signature to key based only.
All signer implementations must be thread-safe.

IVerifier
This processor is the reverse of the ISigner one. Here we take a signature and verify that it is authentic.
All verifier implementations must be thread-safe.

IReferenceLoader
Since references might have to be loaded from a number for location and from different formats this interface represents a contract for such loading. Since all the references will be given in form of a URI it is possible that in the future we might need loaders that are quite sophisticated. All loaders will return string data, which will then be used to sign the reference itself.

ICanonicalizer
This is a contract for creating a canonical form from some input data. Different data streams with the same XML information set may have different textual representations, e.g. differing as to white space. To help prevent inaccurate verification results, XML information sets must first be canonized (standardized) before extracting their bit representation for signature processing.
All canonicalizer implementations must be thread-safe.

IKeyInfoProvider
This is a very special processor in the sense that it will provide key info data. Thus we could implement for example an X509CerificateKeyInfoProvider, which would be able to provide certificate (public key) information. Please note that unlike other processors such as Digesters or Transformers a KeyInfoProvider cannot be an arbitrary type pf processor. It must be one of a couple recognized ones as shown in this DTD excerpt:

```
<!ELEMENT KeyInfo
        (#PCDATA|KeyName|KeyValue|RetrievalMethod|
         X509Data|PGPData|SPKIData|MgmtData %KeyInfo.ANY;)* >
<!ATTLIST KeyInfo Id  ID #IMPLIED >
```

This will be provided as a type of implementation in the form of an enumeration, which will then mark an `IKeyInfoProvider` as being a specific type of one of those providers. This would be the `KeyInfoProviderType` enumeration.
Implementation must be thread-safe.

KeyInfoProviderType
This is an enumeration used to specify available key info provider types that are recognized by this component. These are: `KeyName`, `KeyValue`, `RetrievalMethod`, `X509Data`, `PGPData`, `SKIPData`, and `MgmtData`. This will be used by the SignatureManager (actually by `ISigner` implementations) to identify the correct type of provider for the specific needs at hand.
As an enum it is thread-safe.

KeyValueType
This is a sub-enumeration of Key Values (as defined in `KeyInfoProviderType`) and lists the two currently recognized key value types. RSA and DSA.
As an enum it is thread-safe.

IXmlSignatureProcessor
This is a general (serialization-like) processor contract. It states that implementations must produce an `XmlNode` based on their processing output. Thus for example when we call some `KeyInfoProviderType` its processing method must actually produce the following key info in an xml format for example as here:

```
<KeyInfo>
   <X509Data>
```

```
        <X509SubjectName>CN=Ed Simon,O=XMLSec Inc.,ST=OTTAWA,C=CA</X509SubjectName>
        <X509Certificate> MIID5jCCA0+gA...lVN </X509Certificate>
      </X509Data>
    </KeyInfo>
```

This is achieved through `GetXml()` method which returns a specific xml snippet for the particular processor.

1.5.2 TopCoder.Security.Crytography.Mobile.Digesters

This package provides some well-known and useful digesters.

SHA1Digester

This is an implementation of IDigester interface and provides a SHA-1 digester which provides a 160 bit digest from the input data.
This is a thread-safe implementation, as it has no mutable state.

1.5.3 TopCoder.Security.Crytography.Mobile.Canonicalizers

This will provide simple implementations that will canonize a set of input bytes.

StandardFormCanonicalizer

This is a simple, UTF-8 based, implementation of the ICanonicalizer interface contract. XML documents are usually lexically loose meaning that the same document in terms of data can actually be written with many different sets of bytes even when using the same encoding. For example extra white spaces could be present in one and not the other and of course the order of attributes or even tags could be different.
This implementation will provide a normalized lexical form for XML where all of the allowed variations have been removed, and strict rules are imposed to allow consistent byte-by-byte comparison.
This is thread-safe as there is no state currently provided.

1.5.4 TopCoder.Security.Crytography.Mobile.ReferenceLoaders

WebBasedReferenceLoader

This is a simple implementation of the ICanonicalizer interface, which simply loads the whole resource, specified in the provided URI using the http protocol. Since streams are tricky when it comes to synchronization we lock the Load method on the instance of the class to make it completely thread safe. Note that currently no credentials are being set for access to references.

SoapMessageReferenceLoader

This is a reference loader which will load data from a Soap Envelope (i.e. xml) The expectation of the URI is that it will be of the "#reference"

1.5.5 TopCoder.Security.Crytography.Mobile.Signers

This is the heart of this component is the process that will sign the contents of the document.

RSADSASigner

This is a specific ISigner implementation, which uses RSA and DSA asymmetric encryption algorithms for signing and verifying the signature. This implementation is thread-safe since it has no mutable state.

1.6 Component Exception Definitions

1.6.1 Custom Exceptions

There are a number of things that could go wrong in such a component. From configuration to actual digesting or signing or verifying, there are a number of exceptions that have been abstracted out:

SignatureManagerException
This is a general exception, which deals with issues encountered by the main manager while signing or verifying.

ReferenceLoadingException
This is an exception specific to Reference Loaders and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with loading a reference. This could be for example due to an IO exception.

DigesterException
This is an exception specific to Digesters and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with producing a digest.

TranformerException
This is an exception specific to Transformers and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with processing a transformation of a reference.

CanonicalizationException
This is an exception specific to the process of cononicalization and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with processing a cononicalization of a set of bytes.

SigningException
This is an exception specific to the process of digital signing and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with processing a document to produce its signature.

VerificationException
This is an exception specific to the process of digital signature verification and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with processing a document to verify its signature.

VerificationFailedException
This is the exception used to signal to the user that the verification of the provided signature has failed.

ProcessRegistryException
This is an exception that is thrown if there is an issue with the Registry. This would occur when there are issues with reading or configuring the registry.

KeyInfoException
This is an exception specific to the process of dealing with key info and would normally be caught by the manager and then chained in a SignatureManagerException or its descendant.
It is thrown if there is an issue with processing a key info portion of signing/verifying and could be as an example a case of missing key information or an invalid certificate for example.

1.6.2 System and general exceptions

Here the system exception that we will use.
System.ArgumentException:
This exception is used for invalid arguments. Invalid arguments in this design are empty strings for keys for example or an incorrect value range for a strict value range definition. The exact details are documented for each method in its documentation.

System.ArgumentNullException:
This exception is used in all classes for null arguments and arrays/lists with null elements. The exact details are documented for each method in its documentation.

System.ConfigurationException
We will reuse this exception to provide information about a configuration problem when dealing with ProcessRegistry configuration.

1.7 Thread Safety

While there is no requirement to make this thread-safe it is found to be quite useful to implement this as much to be thread-safe as possible.
Thread-safety of the manager is implemented in the manner where any state that the manager has is immutable (i.e. read-only) and in the manner in which signing and verifying is done: as a utility call with no intermediate state. The idea is that a call to sign a set of references is done in a single method, which means that the call is stateless. One caveat to this would be the fact that a mutable `Reference` value object (or DTO – data transfer object) would allow another (buggy or malicious) thread to modify its state while the manager is working on it. Locking the specific set property method would not help since the issue is not a race condition for a single field but rather an ACID like problem with data being changed from its original state between the micro-managing calls of the signer (i.e. while the signer is digesting some other thread changes the key for encryption for example which has a side effect when the signer gets to the point of encrypting the digest) to deal with this issue it was decided to make the `IReference` implementations immutable and thus completely thread-safe.
All other constituent part of this component (i.e. Digesters, Transformers, etc…) are required to be thread-safe.

2. Environment Requirements

2.1 Environment

- Development language: C#
- Compile target: .Net Compact Framework 2.0

2.2 TopCoder Software Components

- **Compact Configuration Manager 1.0**
  Used to configure elements needed for the processing of the signatures and veririfcations.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

TopCoder.Security.Crytography.Mobile
TopCoder.Security.Crytography.Mobile.Digesters
TopCoder.Security.Crytography.Mobile.Signers
TopCoder.Security.Crytography.Mobile.Canonicalizers
TopCoder.Security.Crytography.Mobile.ReferenceLoaders

3.2 Configuration Parameters

We need to be able to configure different objects for the registry. Which will be done as follows:

Parameter	Description	Values
digesters	These are 1 or more IDigester implementations that map to a specific class name used to create this object. These comma delimited values where the key is the xml based method name in the <DigestMethod Algorithm=name> element. ***Required***	"http://www.w3.org/2000/09/xmldsig#sha1, TopCoder.MyFigester.Sha1"
transformers	These are 0 or more ITransformer implementations that map to a specific class name used to create this object. These comma delimited values where the key is the xml based method name in the <Transform Algorithm=name> element. ***Optional***	"some_name, class.name"
signers	These are 1 or more ISigner implementations that map to a specific class name used to create this object. These comma delimited values where the key is the xml based method name in the <SignatureMethod Algorithm=name> element. ***Required***	"http://www.w3.org/2000/09/xmldsig#dsa-sha1, sha_dsa_signer.class.name"
canonicalizers	These are 1 or more ICanonicalizer implementations that map to a specific class name string used to create this object. These comma delimited values where the key is the xml based method name in the <CanonicalizationMethod Algorithm=name> element. ***Required***	"http://www.w3.org/TR/2001/REC-xml-c14n-20010315, simple_canonical_form.class.name"
reference_loaders	These are 1 or more IReferenceLoader implementations that map to a specific class name used to create this object. These comma delimited values where the key is the protocol that this loader understands and the value is the Object Factory configuration token that we will used to instantiate it. ***Required***	"http, HTTP_uri_loader.class.name"

key_info_providers	These are 1 or more IKeyInfoProvider implementations that map to a specific class name used to create this object. These are comma-delimited values where the key is the specific type of keyinfo and the value is the Object Factory configuration token that we will used to instantiate it. ***Required***	"X509Data, X509Data_Provider.class.name"

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'nant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

None

4.3 Demo

The demo will simulate the signing and the verification of the following type of a document. Here we will assume that we have the following references:

```
"http://www.abccompany.com/index.html" would reference an HTML page on the Web
"http://www.abccompany.com/logo.gif" would reference a GIF image on the Web
"http://www.abccompany.com/xml/po.xml" would reference an XML file on the Web
```

We will also assume that following has been set up in the registry:

Key	Class
"http://www.w3.org/TR/2001/REC-xml-c14n-20010315"	TopCoder.Security.Crytography.Mobile.Canonicalizers.StandardFormCanonicalizer
"http://www.w3.org/2000/09/xmldsig#sha1"	TopCoder.Security.Crytography.Mobile.Digesters.SHA1Digester
"xml:dig:signer:rsa-dsa"	TopCoder.Security.Crytography.Mobile.Signers.RSADSASigner
"xml:dig:transformer:dummy"	TopCoder.Security.Crytography.Mobile.Transformers.DummyTransformer
"http"	TopCoder.Security.Crytography.Mobile.ReferenceLoader.WebBasedReferenceLoader
"X509Data"	Some future implementation of IVerifier
"X509Data"	Some future implementation of IKeyInfoProvider

Please note that the provided transformer is not part of this design and is just a dummy here. The developer can create a simple transformer that will just remove all white space from a reference data.

4.3.1 Signing the document references

```
//
// We will first collect all the necessary data and put it into reference value objects

// 1. We create the ordered list of transformer keys for this resource
// Note that since this transformer has no parameters we pass in an empty dictionary.
IList<InstantiationVO> transformerInstanceDefinitions = new ArrayList();
transformerInstanceDefinitions.Add(new InstantiationVO("xml:dig:transformer:dummy"
                                                       , new Hashtable());

//2. we create the reference with all the data necessary
IReference reference_1 =
              new Reference("http://www.abccompany.com/index.html"        // URI
                            , transformerInstanceDefinitions              // transformers
                            , new InstantiationVO("http://www.w3.org/2000/09/xmldsig#sha1"
                                                  , new Hashtable())      // digester
```

```
                                );
// Here we repeat the above steps and create the rest of the references but this time without
// the transformers (which are optional)
. . .

// We then place all the references in a list:
IList<IReference> references = new ArrayList();
references.Add(reference_1);
references.Add(reference_2);
references.Add(reference_3);
//
// Next we will call the SignatureManager with the necessary data to get this signed

// 1. Create a manager first, which will load the registry as well with necessary keys
SignatureManager manager = new SignatureManager("namespace");

// 2. sign the references, please note that the KeyInfoProvider is not defined in this component
// as an implementation so we will just assume a DSA based key definition
// Please consult http://www.w3.org/2000/09/xmldsig#DSAKeyValue for more information

// create a canonicalizer instance definition with no parameters
IDictionary canonicalizerParams = new Hashtable();
InstantiationVO canonicalizer =
       new InstantiationVO("http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
                           , canonicalizerParams);

// create a signer instance definition with two parameters
IDictionary signerParems = new Hashtable();
signerParams.Add("DSAKeyInfo", … some parameters);
signerParams.Add("RSAKeyInfo", … some parameters);
InstantiationVO signer =
       new InstantiationVO("xml:dig:signer:rsa-dsa", signerParams);


XmlNode xml = manager.Sign(references                          // references to sign
              , canonicalizer                                  // canonicalizer to use
              , signer                                         // signer
              , keyInfoProvider                                // key information, not part
                                                               // of this component
              , "my first signature"                           // Signature id
              );

// At this point we will have a signer document that should look something like this
// please note that the actual signatures and digests are not correct here
 <Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
     <SignedInfo>
         <CanonicalizationMethod  Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
         <SignatureMethod Algorithm="xml:dig:signer:rsa-dsa/>
         <Reference URI="http://www.abccompany.com/index.html"/>
             <Transforms>
                 <Transform Algorithm=" xml:dig:transformer:dummy"/>
             </Transforms>
             <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
             <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
         </Reference>
             <Reference URI=" http://www.abccompany.com/logo.gif">
             <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
             <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
         </Reference>
             <Reference URI=" http://www.abccompany.com/xml/po.xml"/>
             <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
             <DigestValue>j6lwx3rvEPO0vKtMupGhystfaytj86Tr</DigestValue>
         </Reference>
     </SignedInfo>
     <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
     <KeyInfo>
         <KeyValue>
             <DSAKeyValue>
                 <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
             </DSAKeyValue>
         </KeyValue>
```

This functionality will be defined in a different component. Here we are not responsible for generation of this.

```
    </KeyInfo>
  </Signature>
```

4.3.2 Verifying a document

```
//
// we simply accept the node to verify it

try{
       manager.VerifySignature(xmlNode);
}catch(VerificationFailedException vfe){
       . . . // do something with it
}
```

UPDATED DEMO:

A SIMPLE SIGN PROCESS:

```
            //Setup References
            IList<InstantiationVO> transformersList1 = new
List<InstantiationVO>();
            InstantiationVO digester1 = new
InstantiationVO("http://www.w3.org/2000/09/xmldsig#sha1",
                new Dictionary<string, object>());
            IReference reference1 = new Reference("http://www.google.com",
transformersList1, digester1, "http");
            IList<IReference> references = new List<IReference>();
            references.Add(reference1);

            //Setup Canonicalizer
            IDictionary<string, object> c14nParams = new Dictionary<string,
object>();
            InstantiationVO c14nIVO = new
InstantiationVO("http://www.w3.org/TR/2001/REC-xml-c14n-20010315",
                c14nParams);

            //Setup Signers
            DSACryptoServiceProvider dsa = new DSACryptoServiceProvider();
            DSAParameters dsaParams = dsa.ExportParameters(true);
            IDictionary<string, object> signerParams = new Dictionary<string,
object>();
            signerParams.Add("DSAKeyInfo", dsaParams);
            InstantiationVO signerIVO = new
InstantiationVO("xml:dig:signer:rsa-dsa", signerParams);

            //Setup main digester
            InstantiationVO digesterMain = new
InstantiationVO("http://www.w3.org/2000/09/xmldsig#sha1",
                new Dictionary<string, object>());

            SignatureManager sm = new SignatureManager();

            //Sign the references
            XmlNode signed = sm.Sign(references, c14nIVO, signerIVO,
digesterMain, "myFirstSign");
```

A SIMPLE VERIFY PROCESS:

```
            //Now verify the XmlSignature using the VerifySignature method
exposed by the SignatureManager class
            IDictionary<string, object> keyInfoProviderParams = new
Dictionary<string, object>();

            //Note here the use of false only exports the public key to the
DSA parameters
            DSAParameters verificationDSAParams =
dsa.ExportParameters(false);

            //Set the Public Key of the KeyInfoProvider class
            keyInfoProviderParams.Add("PublicKey", verificationDSAParams);

            //Create Instantiation VO for KeyInfoProvider
            InstantiationVO keyInfoProvider = new
InstantiationVO("testKeyInfoProvider", keyInfoProviderParams);

            //Verify .
            sm.VerifySignature(signed, keyInfoProvider);
```

5. Future Enhancements

Add some useful transformers, encrypter, or signers