Ciphers

# General Description and Features

The Ciphers library contains a number of classes which perform complete cryptographic tasks on data: streams, byte arrays and XML documents. Each of the classes implements some cryptographic scheme and produces or consumes crypto-packages. The crypto-packages contain the encrypted data, as well as information for initializing the ciphers like initialization vectors, salt, encrypted symmetric keys, etc.

The asymmetric encryption (used for encrypting the symmetric keys and the hashes, i.e. producing the signatures) use certificates which provide the public and private keys.

All ciphers implement a handful of simple interfaces (e.g. ICipher, IHasher, IXmlCipher, IXmlSigner). The library uses several facilities for managing and persisting the symmetric encryption keys in use and resolving the cryptographic algorithms. The behaviors of these facilities are also defined by interfaces: (IKeyLocationStrategy, IKeyStorage, ISymmetricAlgorithmFactory, IHashAlgorithmFactory) and have internal default implementations which can easily be replaced.

The implementing classes are designed to be easy to integrate with dependency injection containers implementing a [Common Service Locator](https://commonservicelocator.codeplex.com/) adapter (e.g. Unity, StructureMap, MEF, Spring.NET, and others.) For example, if the facility objects needed by the ciphers are not provided directly in the constructor of the cyphers (passed null value), the library will try to resolve them from a DI container with the help of the common service locator library; and if not found there, it will default to the internal implementations.

# Documentation, Samples and Tests

This document is intended to be more of a general description of the package and its components. For detailed information please refer to the help file or the code – all programming elements are well XML comment-documented.

Good usage samples of the ciphers are the unit tests. Most of the unit test methods are implemented in base generic test classes which the actual test classes inherit from. Usually the concrete test classes are also good examples for instantiation of the cipher classes, and the methods in the generics – for using the interfaces and the extensions.

Before running the unit tests make sure to execute the batch file CreateCertificates.cmd which, as you might expect creates four self-signed certificates used in the unit tests.

*Notice the creation of the second two certificates in the batch file which use SHA256 hashing algorithm. The provider and the type of the provider must be explicitly specified. Otherwise the tests will fail with the cryptic message “Invalid key.”*

# Namespaces and Assemblies

All interfaces and classes are defined and implemented in a single assembly: vm.Aspects.Security.Cryptography.Ciphers. There are two namespaces: vm.Aspects.Security.Cryptography and vm.Aspects.Security.Cryptography.Xml. The former contains the interfaces and classes implementing crypto services on streams and byte arrays and the latter implementations of the same on XML documents.

# Algorithms

The classes from the Ciphers package are implemented by leveraging the cryptographic algorithms implemented by the cryptographic service providers in the .NET base class library – from the namespace System.Security.Cryptography. All concrete algorithms are identified by strings. All classes in the package have some sensible default algorithms but they can easily be changed by specifying other algorithm identifiers. The package has defined a number of string constants which contain the names of the most popular algorithms. The constants can be found in the namespace vm.Aspects.Security.Cryptography.Algorithms. The algorithm names are grouped in several static classes by functional type, e.g. Asymmetric, Signature, Hash, etc. Each one of them has a constant Default which is equal to the accepted default algorithm. For example the class Algorithms.Symmetric has a constant Default which is equal to the constant Algorithms.Symmetric.Aes, which in turn is equal to “AESCryptoServiceProvider”.

If the users would like to specify the algorithms in a DI container compatible with the [Common Service Locator](https://commonservicelocator.codeplex.com/) they can add two string instances to the container with resolution names also defined in the above mentioned static classes. The name of the constants are “ResolveName”, e.g. the value of the constant Algorithms.Hash.ResolveName is “DefaultHash” and the value of the constant Algorithms.Symmetric.ResolveName is “DefaultSymmetricAlgorithm”.

Note that at the moment the library implements only one type of asymmetric encryption – RSA and one type of signature – SHA (1, 256, 384, and 512) + RSA.

# Main Interfaces and Implementing Classes

There are two main groups of interfaces which define the behavior of the most of the classes in the package – one that performs cryptographic tasks on plain data (streams and byte arrays) and the other which performs similar tasks on XML documents. The first groups is in the namespace vm.Aspects.Security.Cryptography and contains the interfaces and the implementations of ICipher and IHasher. The second group is in vm.Aspects.Security.Cryptography.Xml and contains interfaces and implementations of IXmlCipher and IXmlSigner. The methods in ICipher and IXmlCipher are used to protect data for confidentiality, i.e. encrypts and decrypts data. IHasher and IXmlSigner protect data for integrity and possibly authentication, i.e. produces and verifies cryptographically strong hash or signature (encrypted hash) of the data.

There are few other interfaces which serve support functions: symmetric key management, storage and retrieval, hash and symmetric encryption algorithms resolution.

The interface ICypher and its extension ICypherAsync have 3 pairs of methods and one property:

* Encrypt and Decrypt operating on System.Stream derived data containers;
* EncryptAsync and DecryptAsync are the asynchronous versions of the above;
* Encrypt and Decrypt operating on data contained in array of bytes - byte[].
* The property Base64Encoded adds Base64 encoding to the encrypted texts.

Here are a few basic samples for using the interfaces:

using System.IO;

using vm.Aspects.Security.Cryptography;

void EncryptStream(Stream dataStream, Stream encryptedStream)

{

    using (ICipher cipher = GetCipher())

        cipher.Encrypt(dataStream, encryptedStream);

}

async void DecryptStreamAsync(Stream encryptedStream, Stream dataStream)

{

    using (ICipher cipher = GetCipher())

        await cipher.Decrypt(encryptedStream, dataStream);

}

byte[] HashArray(byte[] data)

{

    using (IHasher hasher = GetHasher())

        return hasher.Hash(data);

}

public void EncryptXmlDocument()

{

    var document = LoadXml(xmlOrder);

    GetCipher().Encrypt(document, "/order/billing");

}

The class ICipherExtensions adds a few of convenient extensions for encryption and decryption of text (System.String). Similarly IHasherExtensions adds a few extension methods which compute a hash on a string and extensions named VerifyHash which in a case of incorrect hash throw exceptions instead of returning Boolean value, as do the interfaces.

## ICipher

The Ciphers library contains several implementations of ICipher.

Each of the Encrypt methods produces a “crypto-package”. The crypto-package contains the encrypted text as well as zero or more encryption artifacts needed to encrypt the package correctly: e.g. the length and the contents of the initialization vector, the length and the data of the encrypted symmetric key, the length and the data of the hash or the signature.

The property Base64Encoded controls whether to encode the final package with BASE 64 transformation. Note that the encoding does not produce string output – the result is still either Stream object or byte array, however the caller can convert that to string by using the methods of the class System.Text.Encoding.

The Decrypt methods expect the input data to contain a crypto-package produced by the same type of cipher. E.g. the method EncryptedKeyCipher.Encrypt produces a crypto-package that should be decrypted by the method EncryptedKeyCipher.Decrypt or EncryptedKeyCipher.DecryptAsync. The format of each crypto-package is documented in the respective class XML comment-document.

### NullCipher

This is a trivial implementation of ICipher: it does not encrypt or decrypt the input data – it simply copies it clear text to the output destination. This can be very useful during the development process to verify what exactly is being encrypted and decrypted: e.g. to make sure that the PII data is being inserted in and read from the right columns of the database. It can be used also in unit-test scenarios as a mock object. The class has a single default constructor:

ICipher GetCipher()

{

    return new NullCipher();

}

### DpapiCipher

This cipher leverages DPAPI for encrypting and decrypting the data. Therefore it is suitable for programs that are supposed to run only on a single system: data encrypted on one machine cannot be decrypted on a different machine. This is the fastest cipher but with limited application.

### ProtectedKeyCipher

This class and the classes derived from it use some implementation of a symmetric encryption algorithm (by default AES) to protect the data. The symmetric key is encrypted and stored to a named storage location (e.g. a file). The ProtectedKeyCipher class encrypts the symmetric key using DPAPI. In order for more entities to confidentially exchange documents (e.g. a farm of web servers storing and retrieving data to and from a common database), the generated symmetric key needs to be managed appropriately:

* Generate a new key and store it in a key file.
* Backup the key file.
* Delete the key file.
* Export the key from the key file in clear text.
* Import the key to different security contexts (e.g. machine or account).

These tasks are facilitated by the cipher’s behavior defined by the interface IKeyManagement (see below). The provided utility ProtectedKey leverages this interface to facilitate key management operations (see below).

ProtectedKeyCipher is the base class of all remaining non-XML ciphers. It defines the process of encryption, decryption and building the crypto-package in several methods following the GoF design pattern of “method template” which invokes a set of virtual methods whose implementation depends on the concrete class.

The ProtectedKeyCipher has a single constructor where all parameters have default values. Here is how to instantiate an object from this class which uses the triple DES encryption instead of the default AES:

ICipher GetCipher()

{

    return new ProtectedKeyCipher(Algorithms.Symmetric.TripleDes);

}

### PasswordProtectedKeyCipher

This cipher derives the symmetric key from a passed-in password. The derivation is intentionally slow process and this cipher should not be used for frequent encryption and decryption with different passwords. Since the key is derived from the password each time, the cipher does not need to manage the symmetric key and it disables the inherited methods of the IKeyManagement by doing nothing and returning null-s. The password is passed to the cipher in the constructors either as a SecureString object (recommended) or as a plain old string. Internally the password is copied to a SecureString and disposed after the generation of the symmetric key in the first call to the encryption or decryption methods.

The constructor of the cipher takes two additional parameters needed for the password key derivation: number of iterations and salt length. The minimum number of iterations is 4096. Any number below this will result in throwing an exception. The default value is 16384. The more iterations are specified the safer and the slower the key-derivation process is. The minimum salt length is 8 bytes, the default length is 24, and the recommended length is as long as the length of the generated key is, e.g. if the key is 256 bits long the salt should be 32 bytes long.

### EncryptedKeyCipher

This cipher is very similar to ProtectedKeyCipher. The difference is that the symmetric encryption key is protected by encrypting and decrypting it with asymmetric keys stored in a certificate specified in the constructor of the cipher. If the certificate is null the library will try to resolve its instance from the service locator with resolve name “EncryptingCertificate”. This significantly simplifies the management of the key. In order to obtain the symmetric key from the file, an attacker has to have access to the certificate containing the private key. Provided proper certificate management, the key file is well protected and can freely be copied, e-mailed, etc. However because of the overhead for asymmetric encryption and certificate retrieval the initialization of EncryptedKeyCipher is a bit slower than the ProtectedKeyCipher. There is a utility for managing the symmetric key files – EncryptedKey with similar functionality and syntax.

Common problem of the last two ciphers is that if the key file is lost or somehow the key is compromised, all protected documents will be either inaccessible or will lose their confidentiality. The problem is addressed by the EncryptedNewKeyCipher naturally with a certain performance degradation.

The following code loads a certificate from the current user’s personal certificates issued to the subject “vm.EncryptionCipherUnitTest” and passes it to the constructor to be used for encrypting and decrypting the symmetric key:

using System;

using System.Diagnostics;

using System.IO;

using System.Security.Cryptography.X509Certificates;

using vm.Aspects.Security.Cryptography;

ICipher GetCipher()

{

    var store = new X509Store(StoreName.My, StoreLocation.CurrentUser);

    store.Open(OpenFlags.ReadOnly);

    try

    {

        var certs = store.Certificates

                        .Find(X509FindType.FindByTimeValid,DateTime.Now,false)

                        .Find(X509FindType.FindBySubjectName,

"vm.EncryptionCipherUnitTest", false);

        var cert = certs.Count > 0 ? certs[0] : null;

        return new EncryptedKeyCipher(cert,

Symmetric.Default);

    }

    finally

    {

        store.Close();

    }

}

### EncryptedNewKeyCipher

This cipher generates a new symmetric encryption key for each document, encrypts the key with a public key from a certificate and stores it in the crypto-package itself. There is no key file, and no key management. If a key is compromised only the document from the same crypto-package is compromised. The drawback is that the entire encryption process is slower.

In the sample above if you simply replace EncryptedKeyCipher with EncryptedNewKeyCipher you will get a valid working example of instantiating the latter. Since this cipher stores the key in the document the inherited methods of the interface IKeyManagement do nothing and return null-s.

### EncryptedNewKeyHashedCipher

This cipher adds in the crypto-package the cryptographic hash of the encrypted document. This ensures that the included document has not been modified, guaranteeing the integrity of the document. The hash algorithm is specified in the constructor along with the length of the hash “salt” eliminating the possibility of hash dictionary attacks.

The constructor of the cipher has a parameter which specifies the hash algorithm. By default the hash algorithm uses SHA256. In this example we use MD5 for hashing and 3DES for symmetric encryption instead:

new EncryptedKeyCipher(cert, Algorithms.Hash.MD5, Symmetric.TripleDes);

The problem with this cipher is that it does not authenticate the source of the document and therefore is somewhat prone to a man-in-the-middle attack: if the encryption certificate is compromised, so is the hash.

Note that the property Base64Encoded is not supported here and setting it to true will throw InvalidOperationException.

### EncryptedNewKeySignedCipher

This cipher replaces the hash from the above class with cryptographic signature which ensures not only the integrity of the document but also the identity of the source. The cipher requires a second (signing) certificate as a parameter in its constructor. Similarly, if the certificate is null the library will try to resolve its instance from the service locator with resolve name “SigningCertificate”.

**Note that for signing, the cypher supports only the RSA algorithm.**

Below is a sample instantiation where the signature is created from SHA256 hash (the default is SHA1) and the symmetric encryption algorithm is Rijndael (the default is AES):

ICipher GetCipher()

{

    var store = new X509Store(StoreName.My, StoreLocation.CurrentUser);

    store.Open(OpenFlags.ReadOnly);

    try

    {

        var certs = store.Certificates

                         .Find(X509FindType.FindByTimeValid,

DateTime.Now, false)

                         .Find(X509FindType.FindBySubjectName,

"vm.Sha256EncryptionCipherUnitTest", false);

        var exchangeCert = certs.Count > 0 ? certs[0] : null;

        certs = store.Certificates

                         .Find(X509FindType.FindByTimeValid,

DateTime.Now, false)

                         .Find(X509FindType.FindBySubjectName,

"vm.Sha256SignatureCipherUnitTest", false);

        var signCert = certs.Count > 0 ? certs[0] : null;

        return new EncryptedNewKeySignedCipher(

signCert,

exchangeCert,

Algorithms.Hash.Sha256,

Algorithms.Symmetric.Rijndael);

    }

    finally

    {

        store.Close();

    }

}

Note that the property Base64Encoded is not supported here and setting it to true will throw InvalidOperationException.

## IHasher

The Ciphers package contains three hashers: Hasher, KeyedHasher, PasswordHasher, and RsaSigner. Each of these produces a digest of the input text which later can be verified. For strengthening the hash the interface has a property SaltLength (not used by KeyedHasher and RsaSigner) which value by default is 8 but can be overridden in the objects’ constructors.

### Hasher

The hasher computes and verifies digest of the input text. By default, it uses SHA256 algorithm but this can be changed to any of the constants from the class vm.Aspects.Security.Cryptography.Algorithms.Hash.

Here is how to instantiate the hasher which will produce MD5 hasher with 16 bytes salt:

public override IHasher GetHasher()

{

    return new Hasher(Algorithms.Hash.MD5, 16);

}

### KeyedHasher

The keyed hasher (MAC) computes and verifies digest of the input text with the help of a symmetric key algorithm. By default, it uses HMACSHA256 algorithm but this can be changed to any of the constants from the class vm.Aspects.Security.Cryptography.Algorithms.KeyedHash. The secret key is encrypted with asymmetric algorithm. Retrieving of the key is similar to EncryptedKeyCiper: pass the logical name of the key and the registered or the default key location strategy (IKeyLocationStrategy) will expand the name to a physical name (e.g. file name) and the registered or the default may management (IKeyStorage) will retrieve the encrypted key and in turn the hasher will use the certificate to decrypt it. Use the facility MacKey to export/import/create MAC keys. Alternatively, the hasher implements (IKeyManagement).

Here is how to instantiate the hasher which will produce HMACSHA384 hasher:

public override IHasher GetHasher()

{

    return new Hasher(certificate, Algorithms.KeydHash.HmacSha384, keyFileName);

}

### RsaSigner

The signer is a hasher which encrypts the produced hash with the private key of the document’s source and the verifier decrypts the signature with the public key of the source, thus verifying the identity of the source. Then the verifying code computes the hash and compares it to the decrypted signature which guarantees the integrity of the document.

Currently the only supported asymmetric algorithm for creating signatures is RSA. When creating the signer the caller has the option to select the hash algorithm, which by default is SHA1.

This sample creates a signer which uses SHA256 for underlying hashing:

IHasher GetHasher()

{

    var store = new X509Store(StoreName.My, StoreLocation.CurrentUser);

    store.Open(OpenFlags.ReadOnly);

    try

    {

        var certs = store.Certificates

                         .Find(X509FindType.FindByTimeValid,

DateTime.Now, false)

                         .Find(X509FindType.FindBySubjectName,

"vm.Sha256SignatureCipherUnitTest", false);

        var cert = certs.Count > 0 ? certs[0] : null;

        return new RsaSigner(cert, Algorithms.Hash.Sha256);

    }

    finally

    {

        store.Close();

    }

}

### PasswordHasher

This hasher can be used to hash a password, e.g. for storing it in identity store in accordance with RFC 2898 (PBKF2 based on HMAC/SHA1). In other words this algorithm is used for generating encryption artifacts from a password (a.k.a. password encryption). It is also one of the few officially acceptable methods for hashing of passwords. This is an iterative method, where the generated hash is dependent on the number of iterations, which makes the method also slow but appropriate for password verification. The more iterations are specified the slower is the process however the safer is the hash.

The constructor initializes the underlying object with 3 parameters:

1. Number of iterations (min. 1024, default 16384).
2. Hash length (min. 24 bytes, default 64 bytes).
3. Salt length (min. 8 bytes, default 64). It is recommended that the length of the salt is equal to the length of the produced hash.

If any of the above parameters is below its minimum value the constructor will throw an exception.

# XML Ciphers and Signers

The classes from this group can be used to encrypt and/or sign XML documents or selected elements of those according to several standards specified in <http://www.w3.org/TR/xmlenc-core/> and <http://www.w3.org/TR/xmldsig-core/>.

## IXmlCipher

The interface has only two methods – Encrypt and Decrypt and a property - ContentOnly.

The property ContentOnly specifies whether only the content of the XML elements to be encrypted or the whole elements, including the attributes and the element names.

The method Encrypt has three parameters. The first is the XML document that must be encrypted. The second is an XPath expression which selects the elements that need to be replaced by their encrypted representation and if not specified (or null) the whole document will be encrypted. The third parameter is an optional XmlNamespaceManager object which maps the namespace prefixes used in the XPath expression (if any) to XML namespaces.

The method Decrypt has only one parameter – the encrypted document.

The XML ciphers support only the following symmetric algorithms: DES, 3DES, AES – 128/192/256; the RSA asymmetric algorithm and RSA/SHA – 1/256/384/512 signing.

Here is a sample of an order document:

<?xml version="1.0" encoding="utf-16"?>

<order>

<items>

<item quantity="1">.NET Framework Security</item>

<item quantity="1">Essential XML Quick Reference</item>

</items>

<shipping>

<to>Joe Smith</to>

<street>110 Denny Way</street>

<city>Seattle</city>

<zip>98109</zip>

</shipping>

<billing>

<paymentInfo type="Visa">

<number>0000-0000-0000-0000</number>

<expirationDate>09/15/80</expirationDate>

<billingAddress>

<who>Microsoft Corporation</who>

<street>1 Microsoft Way</street>

<city>Redmond</city>

<zip>98052</zip>

</billingAddress>

</paymentInfo>

</billing>

<billing>

<paymentInfo type="Visa">

<number>1111-1111-1111-1111</number>

<expirationDate>09/15/80</expirationDate>

<billingAddress>

<who>Microsoft Corporation</who>

<street>1 Microsoft Way</street>

<city>Redmond</city>

<zip>98052</zip>

</billingAddress>

</paymentInfo>

</billing>

</order>

And this is the same document with encrypted billing elements:

<?xml version="1.0" encoding="utf-16"?>

<order>

<items>

<item quantity="1">.NET Framework Security</item>

<item quantity="1">Essential XML Quick Reference</item>

</items>

<shipping>

<to>Joe Smith</to>

<street>110 Denny Way</street>

<city>Seattle</city>

<zip>98109</zip>

</shipping>

<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element" xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />

<CipherData>

<CipherValue>7Dx5aMy/Kyz...4YI5Y</CipherValue>

</CipherData>

</EncryptedData>

<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element" xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes256-cbc" />

<CipherData>

<CipherValue>7Dx5aMy/Kyz...SUGR+</CipherValue>

</CipherData>

</EncryptedData>

</order>

### ProtectedKeyXmlCipher

The class implements IxmlCipher and IKeyManagement. It is very similar to the ProtectedKeyCipher: uses symmetric encryption, the symmetric key is protected with DPAPI, the constructor takes the same parameters and defaults. The class uses IKeyStorage and IKeyLocationStrategy for locating and storing the symmetric key.

### EncryptedKeyXmlCipher

The class is similar to EncryptedKeyCipher where the session key is protected by encrypting it with asymmetric key from a certificate.

### EncryptedNewKeyXmlCipher

As expected this is similar to EncryptedKeyXmlCipher: each document is encrypted with its own symmetric key encrypted by a certificate and stored in the crypto-package of the encrypted elements.

### EncryptedNewKeySignedXmlCipher

The cipher adds an enveloped SHA/RSA signature to the document.

## IXmlSigner

The XML signing standard specifies three types of signatures which are defined in the enum SignatureLocation:

* Detached – where the signature is an XML document delivered in a separate document from the signed document.
* Enveloped – where the signature is an element enveloped in the XML document and
* Enveloping – where the signature is an element which envelops the signed element.

The interface has a property SignatureLocation which specifies the type of signature. Since the only supported XML algorithms by the standards is SHA/RSA, there is only one signer class - RsaXmlSigner.

The Boolean property IncludeKeyInfo specifies whether the signature should include the optional information about the signing key. Some consider including this information prone to man-in-the-middle attacks.

The method Sign has four parameters: the XML document; the XPath of the signed element(s) or null to sign the whole document; a namespace manager if the XPath expression contains any prefixes. The last parameter is used only for detached signatures in case the whole document is signed. It must specify the URL of the signed document so that it can be specified in the URI attribute of the Reference element of the signature document.

### RsaXmlSigner

This is the only class which implements the IXmlSigner interface.

# Facility Interfaces and Classes

As mentioned above the ciphers leverage the cryptographic services of the .NET framework. However in order to do their job in providing a complete end-to-end cryptographic services the ciphers must accept or implement some strategies for implementing cross-cutting tasks like:

* Choose symmetric encryption algorithm or hash algorithm.
* Generate, export and import encryption keys.
* Store and retrieve the encryption keys to external media.
* Implement an addressing scheme for the external media.

All of these tasks are abstracted by interfaces. The library allows the user to choose or provide their own concrete implementations otherwise it will use some sensible defaults.

## Algorithm Factories

### ISymmetricAlgorithmFactory

This interface is responsible for providing a concrete implementation of the class SymmetricAlgorithm. The concrete symmetric algorithm can be specified either as a string or as an object. The interface hides the details of the inferring of the concrete algorithm class in the method Initialize and abstracts the details of creating the instance behind the method Create. The library tries to resolve the interface implementation from the [Common Service Locator](http://commonservicelocator.codeplex.com/) and if not successful assumes its own implementation. The internal implementation implements the following strategy for resolving the algorithm:

1. If the caller specified an algorithm name to the method Initialize (coming from the cipher’s constructor parameter), the factory will produce an algorithm by invoking SymmetricAlgorithm.Create with parameter the specified algorithm name. Otherwise,
2. The factory will try to resolve the concrete type of the algorithm by resolving it from the [Common Service Locator](http://commonservicelocator.codeplex.com/). This gives the caller the ability to customize various parameters of the algorithm implementation, e.g. key-size, chaining method, etc. If the type was not resolved,
3. The factory will try to resolve a string instance from the [Common Service Locator](http://commonservicelocator.codeplex.com/) with resolve name “DefaultSymmetricAlgorithm”. The factory will pass that value to the method SymmetricAlgorithm.Create. If not found,
4. The factory will assume the default symmetric algorithm AES and 256 bit long encryption key.

### IHashAlgorithmFactory

Very similar to ISymmetricAlgorithmFactory this interface is responsible for providing a concrete implementation of the class HashAlgorithm. If the user does not provide their own implementation the library will use its internal implementation which works very much like the default implementation of ISymmetricAlgorithmFactory. The default hash algorithm is SHA256.

## Handling keys

The ciphers encrypt the protected text with symmetric algorithm like AES, using session keys which are encrypted using asymmetric algorithms, like RSA. The so encrypted session key must be stored for multiple uses for example in a file or in the message itself. It is clear that in the former case there must be some operational procedures for managing the symmetric (or session) key: the key should be accessible for export, import, checking for existence, etc. If the key is stored in external storage (e.g. file), there must be a mechanism to address the storage (e.g. file name), and also to store and retrieve the key from the so addressed media. These behaviors are defined by the following interfaces.

### IKeyManagement

This interface defines the methods for import and export of the symmetric key and is implemented by the ciphers themselves. These methods are needed mostly for operational purposes. For example to distribute the symmetric key in a web farm or to simply archive it. It is implemented by the ciphers which store the encrypted symmetric key in an external media and leverage the next two facility interfaces.

Additionally the package provides a command line utilities called ProtectedKey and EncryptedKey which help with these tasks for the ProtectedKeyCipher and the EncryptedKeyCipher. For help with the utilities at a command line prompt simply type ProtectedKey or EncryptedKey without parameters.

***Note that the interface* IKeyManagemnt *and the utilities* ProtectedKey *and* EncryptedKey *import and export the keys in clear text. When handling the clear text of the keys please use the same policies and best practices as for handling security artifacts like certificates containing private keys.***

### IKeyStorage

The interface deals with the problems of storing and retrieving the key from external media (e.g. file). The interface defines methods for checking the existence of the key, store and retrieve the encrypted text of the key in the media, and deleting the media containing the key. The store and retrieve operations (GetKey and PutKey) have also asynchronous versions. The caller can provide own object implementing this interface either as a parameter to the cipher constructor (possibly injected with DI container); or can register it in a [Common Service Locator](http://commonservicelocator.codeplex.com/) compatible DI container. If the caller does not provide an object implementing the interface the library will use the internal class KeyFile, which stores and loads the key to a file on the local file system.

### IKeyLocationStrategy

The interface defines a single method GetKeyLocation which converts a logical key name to a physical name of the media location. Also it must implement a strategy to deal with null and empty key name strings. If the caller does not provide an object implementing the interface (in the constructor or in the DI container), the library will use the internal class SymmetricKeyLocationStrategy. The latter implements the following strategy for determining the physical name of the key file:

1. If the caller passes a non-empty string, the method will assume that this is the physical path and filename and will return it as is; otherwise,
2. If the caller passes null or empty string, the class will attempt to retrieve the physical location from the application settings section of the application’s configuration file, entry “symmetricKeyLocation”; and if not found there,
3. The library will return the name of the executable with appended suffix “.KEY”, e.g. MyApplication.exe.KEY.

# Using Dependency Injection Container

In order to resolve the concrete implementations of the above mentioned interfaces the cipher objects use several techniques:

* Look for constructor parameters passed directly (or injected by DI container);
* Attempt to resolve them from a DI container abstracted with the [Common Service Locator](https://commonservicelocator.codeplex.com/) interface.
* Assume some sensible default concrete type and instantiate an object of it.

The following sample code uses the Unity DI container to register and use the cryptographic services of the Ciphers library:

void RegisterCiphers()

{

    X509Certificate2 signingCert;

    X509Certificate2 exchangeCert;

    var store = new X509Store(StoreName.My, StoreLocation.CurrentUser);

    store.Open(OpenFlags.ReadOnly);

    try

    {

        exchangeCert = store.Certificates

                            .Find(X509FindType.FindByTimeValid, DateTime.Now, false)

                            .Find(X509FindType.FindBySubjectName,

"vm.EncryptionCipherUnitTest", false)[0];

        signingCert  = store.Certificates

                            .Find(X509FindType.FindByTimeValid, DateTime.Now, false)

                            .Find(X509FindType.FindBySubjectName,

"vm.Sha256SignatureCipherUnitTest", false)[0];

    }

    finally

    {

        store.Close();

    }

    IUnityContainer currentContainer   = new UnityContainer();

    UnityServiceLocator serviceLocator = new UnityServiceLocator(currentContainer);

    currentContainer

        .RegisterInstance<X509Certificate2>(

Algorithms.Symmetric.CertificateResolveName,

exchangeCert)

        .RegisterInstance<X509Certificate2>(

Algorithms.Hash.CertificateResolveName,

signingCert)

        .RegisterType<SymmetricAlgorithm, AesCryptoServiceProvider>(

                new InjectionFactory(

                    c => new AesCryptoServiceProvider() { KeySize = 128 }))

        .RegisterInstance<string>(

                Algorithms.Hash.ResolveName,

                Algorithms.Hash.Sha256)

        .RegisterType<ICipher, EncryptedKeyCipher>(

                new InjectionFactory(

                    c => new EncryptedKeyCipher()))

        .RegisterType<IHasher, RsaSigner>(

                new InjectionFactory(

                    c => new RsaSigner()))

        .RegisterType<IXmlCipher, EncryptedNewKeySignedXmlCipher>(

                new InjectionFactory(

                    c => new EncryptedNewKeySignedXmlCipher()))

        ;

}

The code above loads and registers one encrypting and one signing certificate; registers the AesCryptoServiceProvider as the symmetric algorithm for encryption with key size of 128 bits; registers SHA256 as the algorithm name for hashing; registers the EncryptedKeyCipher as the ICipher implementation, the RsaSigner as the IHasher signing implementation and the EncryptedNewKeySignedXmlCipher will be used for encrypting and signing XML documents. The above code does not register implementations for IKeyLocationStrategy and IKeyStorage and relies on the default file related implementations. If you would like to use different implementations – register them in the DI container and the library will resolve and use them. It is not a bad idea to explicitly register even the default implementations for improved resolution performance.