# Crypto In .Net Micro Framework

*A developers guide*

Abstract:

The .Net Micro Framework version 4.2 includes an extensible cryptographic framework using the PKCS#11 (or Cryptoki) design. This design enables support for multiple cryptographic tokens and a configurable set of primitives. Since the .Net Micro Framework ships with OpenSSL, the developer already has access to the full set of cryptographic primitives. The porting kit includes several sample platforms (including the emulator) that demonstrate how to use the crypto framework. This document will discuss the layout of the crypto framework and how to use and extend the cryptographic libraries in the Micro Framework.

# 1 Managed Cryptographic Infrastructure in .NetMF

This section will discuss how to use the current cryptographic infrastructure to perform common crypto operations.

## 1.1 Supported Crypto Primitives

The following is a list of the supported cryptographic primitives that will be shipped with .Net Micro Framework v4.2. Extending this list of primitives will be discussed later in this document.

- RSA
- AES
- TripleDES
- DSA
- ECDiffieHellman
- ECDSA
- HMAC
- RNG – Random Number Generator

## 1.2 Deviations from the .Net Framework

There are several differences between the desktop framework and the Micro Framework in regards to cryptography. Mainly the differences stem from two reasons: size limitations; and underlying native code infrastructure. The size limitations should be obvious, but the native code infrastructure is more interesting. The Micro Framework chose to implement an industry standard model for managing cryptographic devices (or tokens). The native code level implements a version of the PKCS#11 framework which is the basis for the extensibility of the crypto primitives. This choice forces the managed code to be slightly different from the desktop. We will discuss these differences later.

### 1.2.1 Size related deviations

The desktop .Net Framework uses some design patterns (like the Abstract Factory) for families of primitives (AesCryptoServiceProvider, AESManaged, etc). However, in the Micro Framework this does not translate very well because we do not intend to have multiple implementations of the same primitive on the same device (for obvious size related reasons). In addition, the desktop implementation of hash algorithms includes a new class for each size of hash function even for the same hash algorithm. In contrast, the Micro Framework only includes the HashAlgorithm base class and uses parameters in the constructor to initialize the hash algorithm type and size. This cuts down on the number of classes required for our framework and thus decreases the size of the assembly.

### 1.2.2 PKCS#11 related deviations

Since the .Net Micro Framework uses PCKS#11 for extensibility there are some changes to the constructors of the crypto primitives. Each of the primitives' constructors can take a Session or a CryptoKey parameter. This change is important because it maps a managed object to a crypto token and identifies the session for which the operations will be performed. Sessions are a way of isolating multiple cryptographic operations. Only one operation of a particular crypto type (encryption, hash, etc.) can be executed at a time per session. In addition, cryptographic objects (like keys) are not shared

between sessions.  If you have multiple threads using cryptographic operations they should use different sessions so that they can perform operations concurrently.

Below is a diagram of the .Net Micro Framework crypto system.

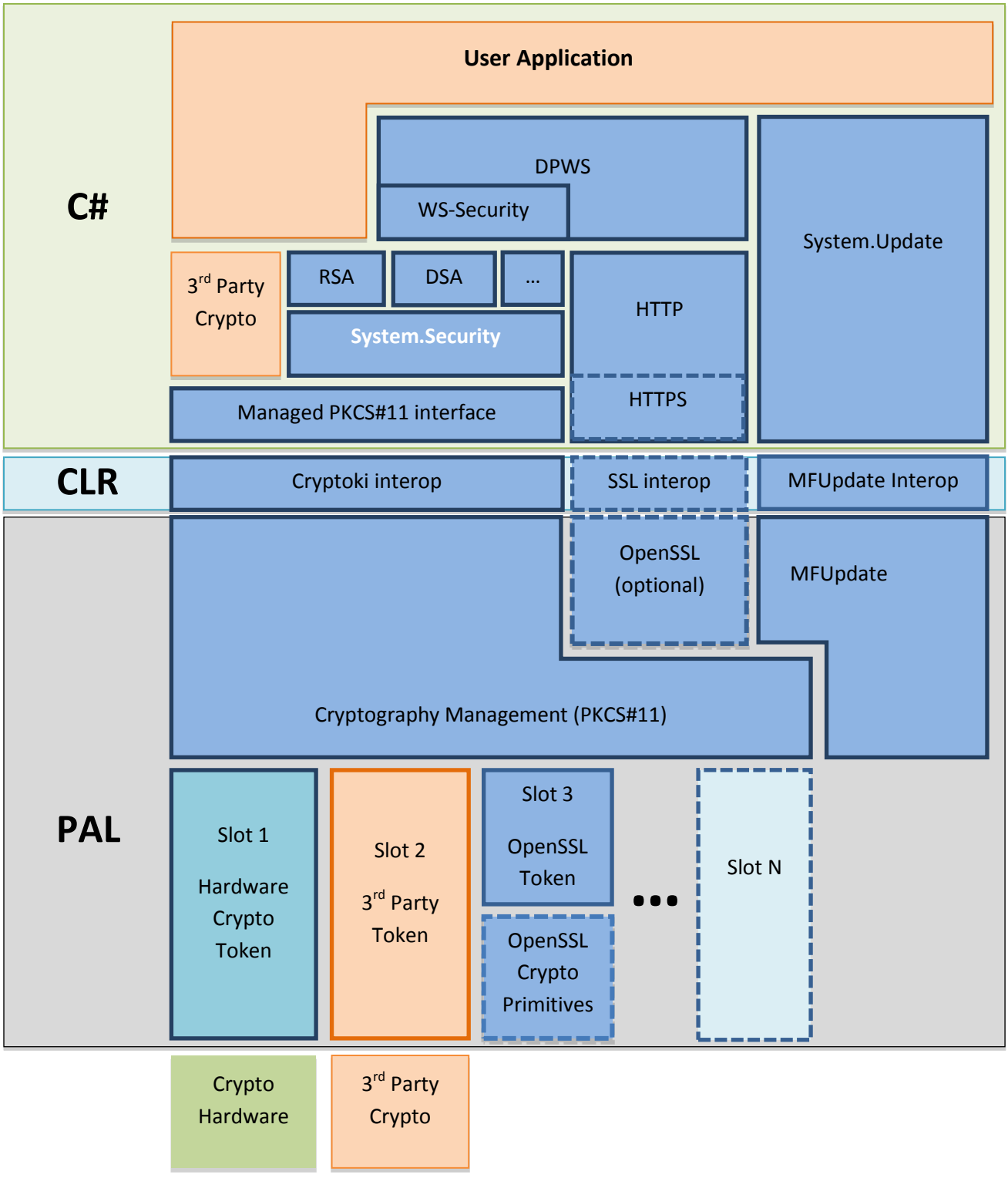


Figure 1: System Diagram

## 1.3 Managed Crypto Framework

The managed crypto framework for .Net Micro Framework is very similar to the Crypto Service Provider classes in the desktop .Net Framework. As described previously, our implementation does not include all of the differentiating subclasses of a particular crypto primitive (like SHA256CryptoServiceProvider). Instead we include the base class and add constructor parameters to distinguish the key length and type of the digest (in this example). For the encryption algorithms, we stripped of the abstract factory pattern because we do not see a reason to support multiple managed implementations of the same primitive. Also all of the constructors have optional parameters for specifying the PKCS#11 session (described later) or a CryptoKey for which the primitive will use for crypto operations. If not specified, a default session will be created for that class. One other main difference between our implementation and the desktop is that we use a key abstraction class rather than the raw bytes for the key. We enable importing of the raw key bytes if required, but we encourage the key generation or key loading to be done at the native (or hardware) levels where it can be better protected if need be.

### 1.3.1 Crypto Service Providers

Supported Algorithms:

- AesCryptoServiceProvider
- DSACryptoServiceProvider
- ECDiffieHellmanCryptoServiceProvider (back ported from ECDiffieHellmanCng)
- ECDsaCryptoServiceProvider (back ported from ECDsaCng)
- RSACryptoServiceProvider
- TripleDESCryptoServiceProvider

### 1.3.2 Hash Algorithms

Supported Hash Algorithms

- SHA1/256/384/512
- RIPEMD160
- MD5

### 1.3.3 HMAC/Keyed Hash Algorithms

Supported Keyed Hash Algorithms:

- HMACSHA1/256/384/512
- HMACMD5
- HMACRIPEMD160

### 1.3.4 Certificates

.Net MF implements a partial X509Certificate2 class with a few modifications. Like the rest of the crypto classes the X509Certificate2 class has a construction parameter to identify the session to be used for creating or loading the certificate. In addition, some of the properties return PKCS#11 objects rather than strings or other .Net Framework objects. Also the public/private key properties return the AsymmetricAlgorithm associated with the key.

The X509Store class enables certificates to be stored and loaded on a given token (if supported on that token). We support "My", "CA", and "Disallowed" stores. The store is also used by OpenSSL when validating a certificate against a root CA.

### 1.3.5 RNG (Random Number Generator)

The random number generator is virtually the same as the desktop version. The only difference being the optional session and service provider constructor parameters to identify the token that will be used to generate the data.

## 1.4 Managed PKCS#11 Framework

The managed PKCS#11 framework is the base framework for the System.Security classes. For users of the .NET Micro Framework these classes (with the exception of the Sessions and CryptoKey classes) can mostly be ignored. That is unless you want to extend the cryptographic framework. Most of the classes in this section are direct representations of the PKCS#11 structures (only in managed code). So for a more in-depth definition of the classes and parameters you should review the PCKS#11 speification.

### 1.4.1 Tokens/Slots

PKCS#11 has the notion of logical slots where cryptographic tokens (or modules) are connected to the framework. A token can be either a physical cryptographic hardware chip or a software implementation. The design allows for multiple crypto providers to be managed within the same framework. It also establishes a common interface for each provider so that providers can be swapped out during build time by Porting Kit users.

The Micro Framework v4.2 will ship with several different sample tokens. The crypto engine included with OpenSSL has been converted into a token for our implementation of PKCS11 as well as our legacy crypto.lib (which is being deprecated). The .Net Micro Framework emulator will have its own token which uses the managed Cng and CryptoServiceProvider classes from the desktop .Net Framework.

The PKCS#11 framework allows you to enumerate the available slots on the device and query for a token's capabilities and general information.

### 1.4.2 Mechanisms

A mechanism is a specific cryptographic operation (such as AES with Cipher Block Chaining). Each mechanism can be associated with a byte array parameter which contains mechanism specific data like the initialization vector (for AES). Each token can be queried for its list of supported mechanisms.

### 1.4.3 Sessions

In order to use a token's cryptographic mechanisms, you first need to establish a session with the token. A session is a logical domain for which cryptographic operations can be performed. Each session maintains its own set of objects (describe next) and state data. This means that you cannot share a key between two sessions without transferring the key via import/export. Also, in our implementation, each session can only handle one operation per crypto type. For example, only one encryption per session can be executing at a time. This goes for encryption, decryption, signing, verification, and hashing (or digesting). PKCS#11 enables several different user states that we currently don't fully support. The PAL

driver will maintain the session states for read-write and login user, but we don't enforce them at this point.

### 1.4.4 Objects

In PKCS#11 there are mainly three types of objects: keys, certificates and data. These items can be loaded, created or searched on a per session basis. The objects in PKCS#11 are similar to xml objects in that they have a set of attributes (or name/value pairs). Both keys and certificates have a set of well-known attribute types that are identified in the managed PCKS#11 object model. These attributes describe the object to the native level implementation which knows how to create or load the object.

### 1.4.5 Crypto Primitives

Nearly all of the crypto primitives (outline in the following subsections) support both a one-shot and multi-part operations. One-shot operations perform a single cryptographic operation on the full chunk of data; whereas, the multi-part operation can perform partial operations on continuous chunks of data.

### 1.4.6 PKCS#11 classes

#### *1.4.6.1 Cryptoki*

Methods:

- SlotList - Gets the list of slots available on the device.
- FindSlots - Finds a slot given the provider name and/or the mechanisms required.

#### *1.4.6.2 Slot*

Methods/Properties:

- Info - Gets the SlotInfo object representing this slot.
- GetTokenInfo - Gets the TokenInfo object representing the crypto token in this slot.
- SupportedMechanisms - Gets the list of supported mechanism types supported by the token in this slot.
- OpenSession - Open a cryptographic session with the token.
- InitializeToken - Initialize the pin and label of an uninitialized token.

#### *1.4.6.3 Mechanism*

Members:

- MechanismType - Type of cryptoki mechanism.
- Parameter - Byte array parameter for the crypto mechanism (like initialization vectors or key handles for the HMACs).

#### *1.4.6.4 Session*

Methods/Properties:

- Handle - Gets the handle of the session in byte[] format.
- Login - Logs into a user role on the token (not currently used).

- Logout - Logs out of a user role on the token (not currently used).
- GetSessionInfo - Gets the session information for this session (state, slotID, flags, device errors).
- InitializePin - Initializes the default token PIN (if token is uninitialized).
- SetPin - Resets the token PIN.
- Close - Closes the session.

#### *1.4.6.5 Object*

Methods/Properties:

- CreateObject - A static method that creates a crypto object in the given session with the given attributes.
- Copy - Makes a copy of a crypto object in the current objects session (not used).
- Save - Saves the object into a persistent storage location on the device (if supported).
- Delete - Deletes the current object from the persistent storage location and the session context.
- GetAttributeValues - Gets the set of attribute values for the given attributes.
- SetAttributeValues - Sets the attribute values for the given set of attributes.
- Size - Gets the size of the current object.
- Dispose - Disposes the object and removes it from the session context.

#### *1.4.6.6 Decryptor*

Methods/Properties:

- ICryptoTransform.InputBlockSize - The input block size for the decryption
- ICryptoTransform.OutputBlockSize - The output block size for the decryption
- ICryptoTransform.CanTransformMultipleBlocks - Determines if the decryptor can perform partial transformations.
- ICryptoTransform.CanReuseTransform - Determines if the decryptor can reuse the decryptor object after completion.
- ICryptoTransform.TransformBlock - Decrypts a block of data as part of a multi-block update.
- ICryptoTransform.TransformFinalBlock - Decrypts the final (or only) block of data.

#### *1.4.6.7 Encryptor*

Methods/Properties:

- ICryptoTransform.InputBlockSize - The input block size for the encryption
- ICryptoTransform.OutputBlockSize - The output block size for the encryption
- ICryptoTransform.CanTransformMultipleBlocks - Determines if the e ncryptor can perform partial transformations.

- ICryptoTransform.CanReuseTransform - Determines if the encryptor can reuse the decryptor object after completion.
- ICryptoTransform.TransformBlock - Encrypts a block of data as part of a multi-block update.
- ICryptoTransform.TransformFinalBlock - Encrypts the final (or only) block of data.

#### *1.4.6.8 CryptokiDigest*

The CryptokiDigest class enables you to create a digest or hash given a set of data. It can also produce a keyed hash or HMAC if a key handle is included in the mechanism parameter of the constructor.

Methods:

- DigestKey - Adds a key to the digest (not used).
- Digest - One shot digest operation
- DigestUpdate - Performa partial digest operation on a chunk of data.
- DigestFinal - Perform the final portion of a partial digest update.

#### *1.4.6.9 CryptokiSign*

Methods/Properties:

- Sign - One-shot signature creation for the given data.
- SignUpdate - Partial signature creation for the given chunk of data.
- SignFinal - Final signature update in a multi-part signature process.

#### *1.4.6.10 CryptokiVerify*

Methods/Properties:

- Verify - One-shot signature verification for the given data.
- VerifyUpdate - Partial signature verification for the given chunk of data.
- VerifyFinal - Final signature verification update in a multi-part signature verification process.

#### *1.4.6.11 CryptokiCertificate*

Methods/Properties:

- GetProperty - Gets the property value for the given property name.
- HasPrivateKey - Determines if the certificate contains a private key.
- CreateCertificate - Static method that creates a cryptoki certificate object given the cryptoki attribute template.
- LoadCertificate - Static method that loads a cryptoki certificate object with the given byte array data and password.
- LoadCertificates - Loads all of the certificates for a given store name.

#### *1.4.6.12 CryptokiRNG*

Methods/Properties:

- GenerateRandom - Generates random bytes to fill the given buffer .
- SeedRandom - Seeds the crypto tokens random number generator.

#### *1.4.6.13 CryptoKey*

Methods/Properties:

- LoadKey - Loads a key given the attribute template.
- Handle - Gets the handle of the key in byte[] format.
- GenerateKey - Generates a key given the mechanism and attribute template.
- GenerateKeyPair - Generates a key pair given the mechanism and attribute templates for the keys.
- DeriveKey - Derives a key based on the given mechanism and the attribute template.
- OpenKey - Opens a key with the given name and key store.
- OpenDeviceKey - Opens the default device key (if supported).
- OpenDeviceAuthorityKey - Opens the default device authority key (if supported). This key is intended to authenticate and verify device data from a server.
- WrapKey - Wraps a key using the given mechanism and key handle.
- UnwrapKey - Unwraps the given key data using the given mechanism, wrapping key.
- ImportKey - Imports a key from a key blob given the key type and class.
- ExportKey - Exports the key to key blob format (if supported by the native code).
- Size - Gets the key size in bits.
- Type - Gets the key type.
- PublicOnly - Determines whether the key only contains public data.

## 1.5 Native Crypto Framework

### 1.5.1 PKCS#11

PKCS#11 is also known as Cryptoki, so you will see the two terms interchanged throughout this document. In the PAL (platform abstraction layer) code we implement a Cryptoki object model that the CLR will interact with. In turn, the cryptoki object model requires HAL (hardware abstraction layer) implementations of for crypto operations. You can think of the PAL layer as a cryptographic token manager rather than doing any specific crypto operations. Each token (software or hardware) implementation has to implement a set or partial set of the following interface structures. Since we do not use inheritance at all in our code base, the interfaces are structs that define a set of function pointers. The token developer does not need to implement every interface nor every method in an interface. To stub a particular interface method the developer simply needs to set the particular function pointer to null. The PAL layer will interpret this as not being supported.

#### *1.5.1.1 ICryptokiToken*

Methods:

- Initialize – Called on system boot or during a CLR reset to initialize the sessions for a token
- Uninitialize – Called during a CLR reset (soft reboot) to uninitialize the sessions for a token
- InitializeToken – Part of the PKCS#11 specification, to initialize a token's pin and label
- GetDeviceError – returns the current device wide error state

#### *1.5.1.2 ICryptokiSession*

Methods:

- OpenSession – Creates a session on the token to enable crypto operations
- CloseSession – Closes the session and any crypto operations in progress
- InitPin – PKCS#11 specification to initialize a user PIN (only for uninitialized tokens)
- SetPin – Resets the user pin, given the old pin
- Login – Logs into a session for the given user type and pin
- Logout – Logs out of the session

#### *1.5.1.3 ICryptokiEncryption*

Methods:

- EncryptInit - Initializes the encryption session with the given mechanism and key
- Encrypt - One shot encryption of the entire chunk of given data
- EncryptUpdate - Partial encryption of a chunk of data
- EncryptFinal - Encrypts the last part of data
- DecryptInit - Initialized the decryption session with given mechanism and key
- Decrypt - One shot decryption of the entire given chunk of data
- DecryptUpdate - Partial decryption of a chunk of data
- DecryptFinal - Decrypts the final portion of data for a partial update

#### *1.5.1.4 ICryptokiDigest*

Methods:

- DigestInit - Initializes the digest (hash) with the given mechanism (for HMACs the key is part of the mechanism parameter)
- Digest - One shot digest for the entire chunk of data
- DigestUpdate - Partial update of the digest
- DigestKey - Digests a key into the hash value (not used in default implementations)
- DigestFinal - Digests the final portion of the partial digest data

#### *1.5.1.5 ICryptokiSignature*

Methods:

- SignInit - Initializes the signature context based on the given mechanism and key
- Sign - One shot signature generation of given data
- SignUpdate - Partial update of the signature process (digest) for given chunk of data
- SignFinal - Finalizes signature of partial update

- VerifyInit - Initializes the signature verification context for given mechanism and key
- Verify - One shot signature verification for given data and signature
- VerifyUpdate - Partial verification update (digest) for the given chunk of data
- VerifyFinal - Final verification of the partial signature check for given signature

#### *1.5.1.6 ICryptokiRandom*

Methods:

- SeedRandom - Seeds the random number generator with given seed bytes
- GenerateRandom - Fills the given parameter with random bytes

#### *1.5.1.7 ICryptokiObject*

Methods:

- CreateObject - Creates a cryptoki object based on the attribute template list
- CopyObject - Copies the given cryptoki object based with given template parameters
- DestroyObject - Destroys the given cryptoki object
- GetObjectSize - Gets the size of the object
- GetAttributeValue - Gets the specified attribute value for the given object
- SetAttributeValue - Sets the specified attribute value for the given object
- FindObjectsInit - Initializes a find objects enumerator
- FindObjects - Finds the given amount of objects matching the supplied criteria
- FindObjectsFinal - Closes the specified find objects enumerator

#### *1.5.1.8 ICryptokiKey*

Methods:

- GenerateKey - Generates a symmetric key for the given mechanism and attributes
- GenerateKeyPair - Generates an asymmetric key pair for the given mechanism
- WrapKey - Wraps a key given the wrapping key and mechanism
- UnwrapKey - Unwraps a key given the wrapping key and mechanism
- DeriveKey - Derives a key from a base key and given attributes
- LoadSecretKey - Loads a secret key from the given key type and data
- LoadRsaKey - Loads an RSA key given the RSA key data
- LoadDsaKey - Loads a DSA key given the DSA key data
- LoadECKey - Loads a EC key given the EC key data

#### *1.5.1.9 ISecureStorage*

The ISecureStorage enables the cryptoki framework to persist crypto objects. While its name implies the storage is secure, it can also be just a plain storage mechanism to store wrapped data (as our samples will show). Currently the .Net MF implements a version of this interface that sits on top of the BlockStorage API.

Methods:

- Create - Creates a new file in the secure storage given entire file data (no partial writes)
- Read - Reads the entire file (no partial reads). Using null as the data parameter should return the size of the file.
- GetFileEnum - Gets a file enumerator for the given file type and group
- GetNextFile - Gets the next file name in the enumeration
- Delete - Deletes the given file

#### *1.5.1.10 CryptokiToken struct*

The CryptokiToken structure identifies a cryptography token implementation in the .Net Micro Framework. Each platform that supports PKCS#11 will have at least one CryptokiToken defined in there platform. The CryptokiToken consists of each of the previous interfaces points in addition with token information, a supported mechanism list, and a few other properties. We provide sample tokens for OpenSSL, the legacy crypto library and for the emulator. However, each platform can either use our sample tokens or create their own. If a particular token does not support some or all of a particular cryptoki interface they can NULL out the interface or specific function pointer and the cryptoki PAL will recognize it as not supported.

Fields:

- TokenInfo - Token information and properties
- TokenWideState - Token wide state data (for all sessions)
- PinWrappingMechansim - Default pin wrapping mechanism
- PinWrappingKey - Default pin wrapping key name (from object store)
- MaxProcessingBytes - Maximum data that should be used for crypto operations. This tells the CLR when to use partial crypto updates to avoid starving other threads.
- MechanismCount - Number of mechanisms supported by this token
- Mechanisms - Pointer to a list of mechanism types
- TokenState - Interface pointer for ICryptokiToken implementation
- Encryption - Interface pointer for ICryptokiEncryption implementation
- Digest - Interface pointer for ICryptokiDigest implementation
- Signature - Interface pointer for ICryptokiSignature implementation
- KeyMgmt - Interface pointer for ICryptokiKey implementation
- ObjectMgmt - Interface pointer for ICryptokiObject implementation
- Random - Interface pointer for ICryptokiRandom implementation
- SessionMgmt - Interface pointer for ICryptokiSession implementation
- Storage - Interface pointer for ISecureStorage implementation

#### *1.5.1.11 CryptokiSession struct*

The CryptokiSession structure represents a session context for a cryptographic operation.

Fields:

- SessionHandle - Session handle for this session

- SlotID - Slot index associated with the session
- ApplicationData - Application data for notify callback (part of PKCS#11 - not implemented)
- Notify - Callback function for application (part of PKCS#11 – not implemented)
- State - Session state data (user type and RO/RW)
- IsLoginContext - Determines if the login type was context specific (PKCS#11)
- Context - Holds context information for each cryptoki interface
- Token - Pointer to the token for which this session is taking place

#### *1.5.1.12 CryptokiSlot struct*

Each platform that supports our Cryptoki framework will be required to declare an array of CryptokiSlot objects that represent the number of tokens supported by the platform. In order to handle pluggable crypto modules (like smart cards), PKCS#11 separates the crypto token (hardware or software) from the dedicated slot which identifies the supported tokens for that slot.

Fields:

- SlotInfo - The slot information that describes which tokens are supported and the tokens availability
- Token - A pointer to the token residing in the slot (or NULL if there is none)

### 1.5.2 OpenSSL

Supported OpenSSL mechanisms (that are enabled by default in Net MF):

- Encryption
  - ECDSA
  - DES3_CBC
  - DES3_CBC_PAD
  - RSA_PKCS
  - AES_CBC
  - AES_CBC_PAD
  - AES_ECB
  - AES_ECB_PAD
- KeyGen
  - EC_KEY_PAIR_GEN
  - ECDH1_DERIVE
  - GENERIC_SECRET_GEN
  - DES3_KEY_GEN
  - DSA_KEY_PAIR_GEN
  - RSA_PKCS_KEY_PAIR_GEN
  - AES_KEY_GEN
- Signing
  - ECDSA

    - DSA
    - RSA_PKCS (with any digest below)
- Digests
    - SHA_1
    - SHA224
    - SHA256
    - SHA384
    - SHA512
    - RIPEMD160
    - MD5
- HMAC
    - SHA_1_HMAC
    - SHA256_HMAC
    - SHA384_HMAC
    - SHA512_HMAC
    - RIPEMD160_HMAC
    - MD5_HMAC

### 1.5.3 Emulator Crypto

Supported emulator crypto mechanisms (that are enabled by default in Net MF):

- Same as OpenSSL except no SHA224

### 1.5.4 Legacy Crypto (deprecated)

Supported Legacy (Crypto.lib) mechanisms:

- Encryption
    - RSA_PKCS
- Signing
    - SHA1_RSA_PKCS
- KeyGen
    - RSA_PKCS_KEY_PAIR_GEN

## 1.6 Extending the Crypto Framework

One of the main goals for the crypto feature was to be able to support a configurable and extendable infrastructure for a variety of different devices. This is partly why we chose to implement a version of the PKCS#11 design. It enables you to easily add new crypto primitives (even proprietary or little known primitives) without having to re-architect the crypto model or add plumbing throughout the code stack. With our implementation, you only need to add a managed class and the corresponding native code implementation. All of the inter-op code is already handled via the PKCS11 object model.

### 1.6.1 Adding More Primitives to an Existing Token

#### *1.6.1.1 Native Code changes*

Adding a new primitive to native code is trivial. Just follow these steps:

- Add the mechanism type to the list of mechanisms in your CryptokiToken declaration
- Add the new primitive in your tokens associated cryptoki interface methods
    - Most cryptoki interfaces have a few methods that need to be updated
        - Initialization (enable your new mechanism here, and setup the context)
        - One shot method (perform the crypto operation based on the context)
        - Partial update method
        - Finalize method (Finish partial update for current context)

#### *1.6.1.2 Managed Code Changes*

The managed code is also fairly trivial in most cases. Follow these steps:

- Adding a new digest
    - Sublcass HashAlgorithm
    - Choose the appropriate MechanismType (or choose a new one) for the HashAlgorithm constructor
    - Override HashSize to return appropriate hash size
- Adding a new Asymmetric Encryption
    - Subclass AsymmetricAlgorithm
    - Assign LegalKeySizes array in constructor
    - Assign default KeySize value in constructor
    - Override GenerateKeyPair() and call CryptoKey.GenerateKeyPair with appropriate attributes for your encryption type
    - Add your own Encryption/Decryption and/or Sign/Verify methods
- Adding a new Symmetric Encryption
    - Subclass SymmetricAlgorithm
    - Set LegalBlockSizesValue and LegalKeySizesValue in constructor
    - Set default BlockSizeValue and KeySize in constructor
    - Override abstract MechanismType property
    - Override abstract CreateDecryptor method
    - Override abstract CreateEncryptor method
    - Override abstract GenerateKey method
    - Override abstract GenerateIV method
    - Override CipherMode property (if needed)
    - Override PaddingMode property (if needed)

### 1.6.2 Adding a New Crypto Token

The easiest way to add a new cryptoki token is to review the existing samples (OpenSSL and the legacy crypto token). These samples are found in the Porting Kit or the CodePlex enlistment in the following folders:

- DeviceCode\PAL\OpenSSL\NetMfPkcsCrypto\
- DeviceCode\PAL\PKCS11\Tokens\Legacy\

## 1.7 Sample Managed Code

For a more complete set of sample code please enlist in our CodePlex project or install the Micro Framework Test MSI (requires our SDK) from the CodePlex download page.

### 1.7.1 Encryption

The following snippet demonstrates asymmetric and symmetric encryption/decryption with AES and RSA.

```
public byte[] AesEncrypt(byte[] data, CryptoKey key)
{
    using (AesCryptoServiceProvider csp = new AesCryptoServiceProvider(key))
    {
        using (ICryptoTransform encr = csp.CreateEncryptor())
        {
            return encr.TransformFinalBlock(data, 0, data.Length);
        }
    }
}

public byte[] AesDecrypt(byte[] encryptedData, CryptoKey key)
{
    using (AesCryptoServiceProvider csp = new AesCryptoServiceProvider(key))
    {
        using (ICryptoTransform decr = csp.CreateDecryptor())
        {
            return decr.TransformFinalBlock(data, 0, data.Length);
        }
    }
}


public byte[] RsaEncrypt(byte[] data, CryptoKey key)
{
    using (RSACryptoServiceProvider csp = new RSACryptoServiceProvider(key))
    {
        return csp.Encrypt(data);
    }
}

public byte[] RsaDecrypt(byte[] encryptedData, CryptoKey key)
{
    using (RSACryptoServiceProvider csp = new RSACryptoServiceProvider(key))
    {
        return csp.Decrypt(encryptedData);
    }
}
```

### 1.7.2 Signing

```
public byte[] DsaSign(byte[] data, CryptoKey key)
{
    using (DSACryptoServiceProvider csp = new DSACryptoServiceProvider(key))
    {
        csp.HashAlgorithm = MechanismType.SHA_1;

        return csp.SignData(data);
    }
}

public bool DsaVerify(byte[] data, byte[] signature, CryptoKey key)
{
    using (DSACryptoServiceProvider csp = new DSACryptoServiceProvider(key))
    {
        csp.HashAlgorithm = MechanismType.SHA_1;

        return csp.VerifySignature(data, signature);
    }
}

public byte[] DsaSignHash(byte[] hashValue, MechanismType hashAlg, CryptoKey key)
{
    using (DSACryptoServiceProvider csp = new DSACryptoServiceProvider(key))
    {
        return csp.SignHash(hashValue, hashAlg);
    }
}

public bool DsaSignHash(byte[] hashValue, MechanismType hashAlg, byte[] signature,
CryptoKey key)
{
    using (DSACryptoServiceProvider csp = new DSACryptoServiceProvider(key))
    {
        return csp.VerifyHash(hashValue, hashAlg, signature);
    }
}
```

### 1.7.3 Digests

```
public byte[] Sha512ComputeHash(byte[] data)
{
    using (HashAlgorithm csp = new HashAlgorithm(HashAlgorithmType.SHA512))
    {
        return csp.ComputeHash(data);
    }
}

public byte[] MD5ComputeHash(byte[] data)
{
    using (HashAlgorithm csp = new HashAlgorithm(HashAlgorithmType.MD5))
    {
```

```
        return csp.ComputeHash(data);
    }
}
```

### 1.7.4 HMAC (KeyedHash)

```
byte[] HmacSha256(byte[] data, CryptoKey key)
{
    using (KeyedHashAlgorithm hmac = new KeyedHashAlgorithm(
KeyedHashAlgorithmType.HMACSHA256, key))
    {
        return hmac.ComputeHash(data);
    }
}

byte[] HmacRipemd160(byte[] data, CryptoKey key)
{
    using (KeyedHashAlgorithm hmac = new KeyedHashAlgorithm(
KeyedHashAlgorithmType.HMACRIPEMD160, key ))
    {
        return hmac.ComputeHash(data);
    }
}
```

### 1.7.5 Key Generation/Loading

```
public CryptoKey AesGenerateKey(Session session)
{
    using (AesCryptoServiceProvider csp = new AesCryptoServiceProvider(session))
    {
        csp.GenerateKey();

        return csp.Key;
    }
}

public CryptoKey RsaGenerateKeyPair(Session session)
{
    using (RSACryptoServiceProvider csp = new RSACryptoServiceProvider(session))
    {
        return csp.KeyPair;
    }
}
```

RSA Key loading from an RSAKeyParameters object:

```
public CryptoKey RsaImportKey(RSAParameters rsaParms, Session session)
{
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(session))
    {
        rsa.ImportParameters(rsaParms);

        return rsa.KeyPair;
    }
}

public RSAParameters RsaExportNewKey(int keySize)
{
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(keySize))
    {
        return rsa.ExportParameters(true);
    }
}
```

RSA key loading from a binary RSAKeyBlob:

```
public CryptoKey RsaImportKey(byte[] rsaKeyBlob, Session session)
{
    return CryptoKey.ImportKey(session, rsaKeyBlob, CryptoKey.KeyClass.Private,
CryptoKey.KeyType.RSA, true);
}
```

RSA key loading from the RSA key elements:

```
public byte[] SignWithLoadedRsaKey(byte[] data, HashAlgorithm alg)
{
    using (Session session = new Session("", MechanismType.RSA_PKCS))
    using (CryptoKey privateKey = CryptoKey.LoadKey(session, m_importKeyPrivate))
    {
        using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(privateKey))
        {
            return rsa.SignData(data, alg);
        }
    }
```

```
}

internal static CryptokiAttribute[] m_importKeyPublic = new CryptokiAttribute[]
{
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Class  ,
Utility.ConvertToBytes((int)CryptokiClass.PUBLIC_KEY)),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.KeyType,
Utility.ConvertToBytes((int)CryptoKey.KeyType.RSA)),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Modulus, new byte[]
        {
            0xC6, 0x29, 0x73, 0xE3, 0xC8, 0xD4, 0xFC, 0xB6,
            0x89, 0x36, 0x46, 0xF9, 0x58, 0xE5, 0xF5, 0xE5,
            0x25, 0xC2, 0xE4, 0x1E, 0xCC, 0xA8, 0xC3, 0xEF,
            0xA2, 0x8D, 0x24, 0xDE, 0xFD, 0x19, 0xDA, 0x08,
            0x46, 0x9A, 0xA9, 0xBA, 0xAE, 0x77, 0x20, 0x28,
            0xED, 0x51, 0x43, 0x8C, 0x28, 0x6F, 0x99, 0x5B,
            0x6B, 0x0C, 0x08, 0x7C, 0x4C, 0x7D, 0x6F, 0xCF,
            0xD0, 0xF0, 0xAC, 0x2A, 0x9B, 0x28, 0x28, 0x62,
            0x52, 0x3F, 0x56, 0x3B, 0x6F, 0x49, 0x10, 0x11,
            0x48, 0x45, 0x36, 0x51, 0x62, 0xAE, 0x8C, 0x66,
            0xE8, 0x53, 0x8D, 0x18, 0xDF, 0x21, 0x12, 0x30,
            0x35, 0x79, 0xAD, 0x41, 0x0F, 0xED, 0x50, 0x41,
            0x26, 0xC3, 0x3E, 0xFE, 0x88, 0xEB, 0xA8, 0x7C,
            0xF2, 0x48, 0x13, 0x84, 0x27, 0xCE, 0x19, 0x86,
            0x33, 0x14, 0x89, 0xEB, 0x7A, 0x90, 0x21, 0x46,
            0x5C, 0xC2, 0x22, 0x23, 0x96, 0x06, 0x85, 0xF7,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.PublicExponent, new byte[]
        {
                0x01, 0x00, 0x01
        }),
};

internal static CryptokiAttribute[] m_importKeyPrivate = new CryptokiAttribute[]
{
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Class  ,
Utility.ConvertToBytes((int)CryptokiClass.PRIVATE_KEY)),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.KeyType,
Utility.ConvertToBytes((int)CryptoKey.KeyType.RSA)),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Modulus, new byte[]
        {
            0xC6, 0x29, 0x73, 0xE3, 0xC8, 0xD4, 0xFC, 0xB6,
            0x89, 0x36, 0x46, 0xF9, 0x58, 0xE5, 0xF5, 0xE5,
            0x25, 0xC2, 0xE4, 0x1E, 0xCC, 0xA8, 0xC3, 0xEF,
            0xA2, 0x8D, 0x24, 0xDE, 0xFD, 0x19, 0xDA, 0x08,
            0x46, 0x9A, 0xA9, 0xBA, 0xAE, 0x77, 0x20, 0x28,
            0xED, 0x51, 0x43, 0x8C, 0x28, 0x6F, 0x99, 0x5B,
            0x6B, 0x0C, 0x08, 0x7C, 0x4C, 0x7D, 0x6F, 0xCF,
            0xD0, 0xF0, 0xAC, 0x2A, 0x9B, 0x28, 0x28, 0x62,
            0x52, 0x3F, 0x56, 0x3B, 0x6F, 0x49, 0x10, 0x11,
            0x48, 0x45, 0x36, 0x51, 0x62, 0xAE, 0x8C, 0x66,
            0xE8, 0x53, 0x8D, 0x18, 0xDF, 0x21, 0x12, 0x30,
            0x35, 0x79, 0xAD, 0x41, 0x0F, 0xED, 0x50, 0x41,
            0x26, 0xC3, 0x3E, 0xFE, 0x88, 0xEB, 0xA8, 0x7C,
            0xF2, 0x48, 0x13, 0x84, 0x27, 0xCE, 0x19, 0x86,
            0x33, 0x14, 0x89, 0xEB, 0x7A, 0x90, 0x21, 0x46,
            0x5C, 0xC2, 0x22, 0x23, 0x96, 0x06, 0x85, 0xF7,
        }),
```

```
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.PublicExponent, new byte[]
        {
            0x01, 0x00, 0x01
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.PrivateExponent, new byte[]
        {
            0x6A, 0xBE, 0x93, 0xAD, 0xE5, 0x56, 0x4E, 0x17,
            0x6A, 0x0C, 0x71, 0xE9, 0x09, 0xA9, 0x3E, 0x6F,
            0x44, 0x8B, 0x1A, 0x65, 0x38, 0xEB, 0xC4, 0x38,
            0x47, 0x00, 0xEF, 0x16, 0xAB, 0x92, 0x8C, 0x6F,
            0x9E, 0xD0, 0xDB, 0x93, 0x33, 0x3D, 0xFA, 0x75,
            0xF1, 0x78, 0xB0, 0x01, 0x45, 0x1A, 0xF0, 0xAA,
            0x5D, 0x1C, 0xAB, 0x49, 0x81, 0xCE, 0xA4, 0x37,
            0x77, 0x1E, 0xDE, 0x2F, 0x49, 0x4B, 0x35, 0x8C,
            0xE5, 0xCB, 0x24, 0xA0, 0x31, 0x51, 0xDB, 0x2B,
            0xF9, 0x16, 0x3A, 0xCA, 0xAB, 0x7B, 0x2A, 0x61,
            0xB6, 0xE7, 0xE1, 0x4E, 0x9B, 0xB3, 0xC2, 0x99,
            0x23, 0x6B, 0xB8, 0x43, 0x97, 0x7B, 0xFD, 0x33,
            0x95, 0x73, 0xA5, 0xC2, 0xCF, 0xCC, 0xC4, 0x27,
            0x30, 0xCD, 0xBC, 0x51, 0x2D, 0xDD, 0x22, 0x89,
            0xF2, 0xA3, 0x93, 0x46, 0x65, 0x84, 0x03, 0x8A,
            0x8F, 0x6C, 0x30, 0xB4, 0xF5, 0x67, 0xC4, 0x81,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Prime1, new byte[]
        {
            0xF3, 0x73, 0x8B, 0x7C, 0xA0, 0x95, 0x28, 0x10,
            0xD3, 0x75, 0x4A, 0x69, 0xC9, 0x7A, 0xCB, 0xE5,
            0x28, 0xBD, 0x9A, 0x36, 0x53, 0xCA, 0x00, 0x1A,
            0x21, 0x56, 0x00, 0x7A, 0xE8, 0xDF, 0xC2, 0x3D,
            0x12, 0x5C, 0xE1, 0x7E, 0x0F, 0xC8, 0xCC, 0x9C,
            0x50, 0x01, 0xF1, 0xAE, 0xC3, 0x3B, 0x7B, 0x01,
            0xA4, 0xAC, 0xAF, 0x77, 0xED, 0xAD, 0x20, 0x56,
            0x77, 0x89, 0x99, 0x39, 0xFC, 0xA9, 0xE7, 0x41,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Prime2, new byte[]
        {
            0xD0, 0x60, 0x4A, 0xE6, 0x18, 0xB4, 0x3C, 0x7B,
            0xAC, 0x0B, 0xA3, 0x78, 0x58, 0x05, 0x00, 0xA3,
            0xD9, 0xA2, 0xD7, 0x24, 0xF5, 0xF6, 0xB0, 0x4C,
            0xE6, 0x62, 0x24, 0x44, 0xF6, 0x25, 0x02, 0x26,
            0xE5, 0x8D, 0xAE, 0x6E, 0xBF, 0x16, 0x57, 0xD7,
            0xA2, 0x94, 0x3C, 0xBE, 0x99, 0x9D, 0x80, 0x34,
            0xB5, 0x68, 0x62, 0x20, 0x96, 0x5C, 0x89, 0xDC,
            0xF8, 0x1E, 0xBD, 0x81, 0xB0, 0xFF, 0x17, 0x37,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Exponent1, new byte[]
        {
            0x42, 0x95, 0x23, 0x5D, 0x1E, 0x7E, 0x2C, 0xCB,
            0x0D, 0x4A, 0x52, 0xE3, 0xC3, 0xDA, 0xF5, 0xD0,
            0xE2, 0xE7, 0x98, 0x39, 0xAB, 0x88, 0xDF, 0xA6,
            0x45, 0xDF, 0xC3, 0x99, 0xD9, 0xFE, 0xF8, 0x9C,
            0xC3, 0x5C, 0xEB, 0xBF, 0x12, 0x8A, 0x14, 0x8B,
            0xDB, 0xC5, 0xEC, 0x57, 0xA3, 0xC5, 0xAC, 0xCA,
            0xB2, 0x43, 0x18, 0x6A, 0x70, 0x72, 0x9D, 0x19,
            0x88, 0xEF, 0xF5, 0x1C, 0x4A, 0xE2, 0x1D, 0x01,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Exponent2, new byte[]
        {
```

```
            0xC5, 0x7A, 0x0C, 0x61, 0x66, 0x16, 0x21, 0x9F,
            0xDE, 0xDB, 0xA4, 0xCF, 0x5F, 0x33, 0x56, 0x78,
            0xF1, 0xBF, 0x76, 0x7F, 0x6B, 0xAE, 0x9F, 0x44,
            0x31, 0xAD, 0xDE, 0xCB, 0x90, 0x2E, 0x60, 0x8C,
            0xB6, 0x4E, 0x00, 0x7A, 0xAA, 0x13, 0xA5, 0xAA,
            0x11, 0x44, 0xC5, 0x10, 0xA9, 0x0A, 0x6F, 0xBF,
            0x04, 0x10, 0xE9, 0xB6, 0x12, 0x69, 0x9E, 0xA9,
            0xD0, 0x67, 0x69, 0x97, 0x68, 0x43, 0x48, 0x1F,
        }),
    new CryptokiAttribute(CryptokiAttribute.CryptokiType.Coefficent, new byte[]
        {
            0xCF, 0x52, 0x93, 0x95, 0xDC, 0xBA, 0x71, 0x56,
            0x13, 0x1C, 0x7A, 0x08, 0x97, 0x11, 0x8E, 0x55,
            0xBC, 0x30, 0xC4, 0xE1, 0xE3, 0x17, 0xC4, 0x94,
            0xB2, 0x6B, 0x7A, 0x2D, 0xBD, 0x1F, 0x2C, 0x6D,
            0x78, 0x34, 0x8F, 0x65, 0x22, 0x97, 0xCD, 0xC4,
            0x02, 0x66, 0x10, 0x61, 0x5A, 0x3E, 0x02, 0xCB,
            0xB6, 0x3A, 0x7A, 0x60, 0xEF, 0xAA, 0xB2, 0xB9,
            0x05, 0x9E, 0x76, 0xC9, 0xED, 0x59, 0x77, 0x16,
        }),
};
```