Microsoft

Non-proprietary Security Policy for

FIPS 140-2 Validation

Cryptographic Primitives Library
(bcryptprimitives.dll and ncryptsslp.dll)
in
Microsoft Windows 10
Windows 10 Pro
Windows 10 Enterprise
Windows 10 Enterprise LTSB
Windows 10 Mobile
Windows Server 2016 Standard
Windows Server 2016 Datacenter
Windows Storage Server 2016

DOCUMENT INFORMATION

Version Number	1.1
Updated On	January 13, 2017

CHANGE HISTORY

Date	Version	Updated By	Change
22 DEC 2016	1.0	Tim Myers	First release to validators
13 JAN 2017	1.1	Tim Myers	Update based on CMVP comments

TABLE OF CONTENTS

1   Introduction

The Microsoft Windows Cryptographic Primitives Library is a general purpose, software-based, cryptographic module. The primitive provider functionality is offered through one cryptographic module comprised of two Dynamic Link Library (DLL) files, BCRYPTPRIMITIVES.DLL and NCRYPTSSLP.DLL, subject to FIPS-140-2 validation. Cryptographic Primitives Library provides cryptographic services, through its documented interfaces, to components and applications running on the Windows operating system.

The cryptographic module, Cryptographic Primitives Library, encapsulates several different cryptographic algorithms in an easy-to-use cryptographic module accessible via the Microsoft CNG (Cryptography, Next Generation) API. It can be dynamically linked into applications by software developers to permit the use of general-purpose FIPS 140-2 Level 1 compliant cryptography.

The Operational Environments (OEs) are:
1. Windows 10 Enterprise Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
2. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Pro 3 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
3. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Pro 4 – Intel Core i5 with AES-NI and PCLMULQDQ and SSSE 3
4. Windows 10 Enterprise Anniversary Update (x64) running on a Microsoft Surface Book – Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
5. Windows 10 Enterprise Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
6. Windows 10 Enterprise Anniversary Update (x64) running on a HP Compaq Pro 6305 - AMD A4 with AES-NI and PCLMULQDQ and SSSE 3
7. Windows 10 Pro Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
8. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Pro 3 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
9. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Pro 4 - Intel Core i5 with AES-NI and PCLMULQDQ and SSSE 3
10. Windows 10 Pro Anniversary Update (x64) running on a Microsoft Surface Book - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
11. Windows 10 Pro Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
12. Windows 10 Anniversary Update (x86) [consumer] running on a Microsoft Surface 3 - Intel Atom x7 with AES-NI and PCLMULQDQ and SSSE 3
13. Windows 10 Anniversary Update (x86) [consumer] running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
14. Windows 10 Anniversary Update (x64) [consumer] running on a Dell XPS 8700 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
15. Windows 10 Enterprise LTSB Anniversary Update (x86) running on a Dell Inspiron 660s - Intel Core i3 without AES-NI or PCLMULQDQ or SSSE 3
16. Windows 10 Enterprise LTSB Anniversary Update (x64) running on a Dell Precision Tower 5810MT - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3

17. Windows 10 Enterprise LTSB Anniversary Update (x64) running on a Dell XPS 8700 - Intel Core i7 with AES-NI and PCLMULQDQ and SSSE 3
18. Windows Server 2016 Standard Edition running on a HP Compaq Pro 6305 - AMD A4 with AES-NI and PCLMULQDQ and SSSE 3
19. Windows Server 2016 Standard Edition running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
20. Windows Server 2016 Datacenter Edition running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
21. Windows Storage Server 2016 running on a Dell PowerEdge R630 Server - Intel Xeon with AES-NI and PCLMULQDQ and SSSE 3
22. Windows 10 Mobile Anniversary Update running on a Microsoft Lumia 950 - Qualcomm Snapdragon 808 (A57, A53)

herein referred to as Windows 10 OEs.

1.1 List of Cryptographic Module Binary Executables

BCRYPTPRIMITIVES.DLL – Version 10.0.14393 for Windows 10 OEs

NCRYPTSSLP.DLL – Version 10.0.14393 for Windows 10 OEs

1.2 Brief Module Description

BCRYPTPRIMITIVES.DLL provides cryptographic primitive services.

NCRYPTSSLP.DLL provides cryptographic key derivation function (KDF) primitives for Internet Key Exchange (IKE) versions 1 and 2 – IKEv1 and IKEv2.

1.3 Validated Platforms

The Cryptographic Primitives Library components listed in Section 1.1 were validated using the machine configurations specified in the list of Windows 10 OEs.

1.4 Cryptographic Boundary

The software binaries that comprises the cryptographic boundary for Cryptographic Primitives Library are BCRYPTPRIMITIVES.DLL and NCRYPTSSLP.DLL. The Crypto boundary is also defined by the enclosure of the computer system on which Cryptographic Primitives Library is to be executed. The physical configuration of Cryptographic Primitives Library, as defined in FIPS-140-2, is multi-chip standalone.

2 Security Policy

Cryptographic Primitives Library operates under several rules that encapsulate its security policy.

- Cryptographic Primitives Library is supported on Windows 10 OEs.
- Cryptographic Primitives Library operates in FIPS mode of operation only when used with the FIPS approved version of Windows 10 OEs Code Integrity (ci.dll) validated to FIPS 140-2 under Cert. # 2935, respectively, operating in FIPS mode. This is required to satisfy crypto module integrity checks (See section 4). Additionally there is a functional dependency on CNG.SYS (Cert. # 2936) operating in FIPS mode, required for entropy input (see section on BCryptGenRandom).

Cryptographic Primitives Library

- Windows 10 OEs are operating systems supporting a "single user" mode where there is only one interactive user during a logon session.
- Cryptographic Primitives Library is only in its Approved mode of operation when Windows is booted normally, meaning Debug mode is disabled and Driver Signing enforcement is enabled.
- Cryptographic Primitives Library operates in its FIPS mode of operation by setting any one of the following DWORD registry values to 1:
  - \HKLM\System\CurrentControlSet\Control\Lsa\FipsAlgorithmPolicy\Enabled
  - \HKLM\System\CurrentControlSet\Control\Lsa\FipsAlgorithmPolicy
  - \HKLM\System\CurrentControlSet\Control\Lsa\FipsAlgorithmPolicy\MDMEnabled
  - \HKLM\SYSTEM\CurrentControlSet\Policies\Microsoft\Cryptography\Configuration\Self TestAlgorithms

  Changes to the FIPS mode registry setting do not take effect until the Windows OS has been rebooted. The registry security policy settings can be observed with the regedit tool to determine whether the module is in FIPS mode.
- Instead of editing the registry directly, the FIPS Local/Group Security Policy Flag may be enabled. The Windows operating system provides a group (or local) security policy setting, "System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing", which when enabled, will in turn enable one of the FIPS mode registry settings listed above. Changes to the FIPS mode security policy setting do not take effect until the Windows OS has been rebooted.
- When properly initialized as per the guidance in this section, the Cryptographic Primitives Library will make the determination that it is validated while under the control of the DllMain code block invoked by the OS Loader (as per FIPS 140-2 IG 9.10). When the determination has been made that the module is validated, the module will operate in its FIPS mode of operation. If the module has not been properly initialized as per the guidance in this section, then the module will make the determination that it is not validated.
- All users assume either the User or Cryptographic Officer roles.
- Cryptographic Primitives Library provides no authentication of users. Roles are assumed implicitly. The authentication provided by the Windows 10 OEs operating systems is not in the scope of the validation.
- All cryptographic services implemented within Cryptographic Primitives Library are available to the User and Cryptographic Officer roles.

The following diagram, Figure 1, illustrates the master components of the Cryptographic Primitives Library module:
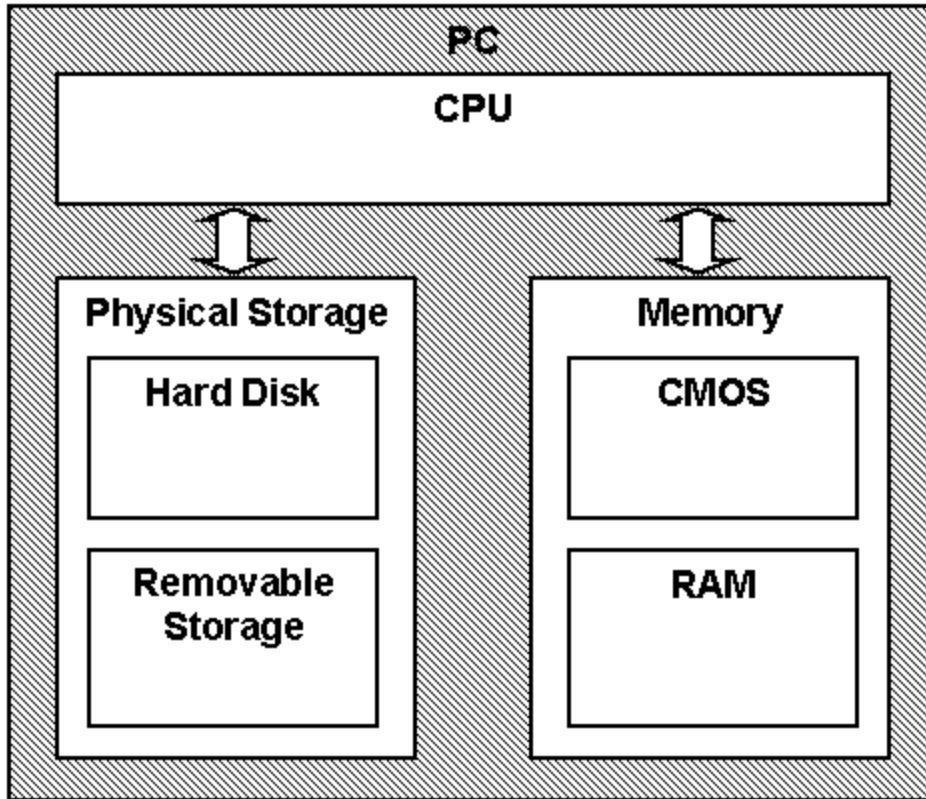
Figure 1 Master Components

2.1 FIPS 140-2 Approved Algorithms

Cryptographic Primitives Library implements the following FIPS-140-2 Approved algorithms:

- FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512 (Cert. # 3347)
- FIPS 198-1 SHA-1, SHA-256, SHA-384, SHA-512 HMAC (Cert. # 2651)
- SP 800-67r1 Triple-DES (2 key legacy-use decryption[1] and 3 key encryption/decryption) in ECB, CBC, CFB8 and CFB64 modes (Cert. # 2227)
- FIPS 197 AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes (Cert. # 4064)
- SP 800-38B and SP 800-38C AES-128, AES-192, and AES-256 in CCM and CMAC modes (Cert. # 4064)
- SP 800-38D AES-128, AES-192, and AES-256 GCM decryption and GMAC (Cert. # 4064)
- SP 800-38E XTS-AES[2] XTS-128 and XTS-256 (Cert. # 4064)
- FIPS 186-4 RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signature generation and verification with 2048 and 3072 modulus; supporting SHA-1[3], SHA-256, SHA-384, and SHA-512 (Cert. # 2193 and # 2192)
- FIPS 186-4 RSA key-pair generation with 2048 and 3072 moduli (Cert. # 2195)

---

[1] Two-key Triple-DES Decryption is only allowed for Legacy-usage (as per SP 800-131A). The use of two-key Triple-DES Encryption is disallowed.
[2] For XTS-AES, as enforced by policy, the length of the data unit shall not exceed 2^20 blocks. XTS-AES mode can only be used for the cryptographic protection of data on storage devices.
[3] SHA-1 is only acceptable for legacy signature verification.

- FIPS 186-4 DSA PQG(gen/ver) and SIG(gen/ver) (Cert. # 1098)
- FIPS 186-4 ECDSA with the following NIST curves: P-256, P-384, P-521 (Cert. # 911)
- KAS – SP 800-56A Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) with parameter FB (p=2048, q=224) and FC (p=2048, q=256); key establishment methodology provides at least 112 bits of encryption strength (Cert. # 92)
- KAS – SP 800-56A EC Diffie-Hellman Key Agreement; Elliptic Curve Cryptography (ECC) with parameter EC (P-256 w/ SHA-256), ED (P-384 w/ SHA-384), and EE (P-521 w/ SHA-512); key establishment methodology provides between 128 and 256-bits of encryption strength (Cert. # 92)
- SP 800-56B RSADP mod 2048 (Cert. # 887)
- SP 800-90A AES-256 counter mode DRBG (Cert. # 1217)
- SP 800-108 Key Derivation Function (KDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512) (Cert. # 101)
- SP 800-132 KDF (also known as PBKDF) with HMAC (SHA-1, SHA-256, SHA-384, SHA-512) as the pseudo-random function (vendor-affirmed)
- SP 800-135 IKEv1, IKEv2, and TLS KDF primitives (Cert. # 886)[4]
- SP 800-38F AES Key Wrapping (128, 192, and 256) (Cert. # 4062)

2.2 Non-Approved Algorithms

- Cryptographic Primitives Library implements the SHA-1 hash, which is disallowed for use in digital signature generation. It can be used for digital signature verification legacy-use. Its use is Acceptable for non-digital signature generation applications.
- If HMAC-SHA1 is used, key sizes less than 112 bits (14 bytes) are not allowed for usage in HMAC generation, as per SP 800-131A.
- Cryptographic Primitives Library implements RSA 1024-bits for digital signature generation, which is disallowed. RSA 2048-bits and 3072-bits are also supported, which are Acceptable.
- Cryptographic Primitives Library supports SP 800-56A Key Agreement using Finite Field Cryptography (FFC) with parameter FA (p=1024, q=160), which is disallowed. The key establishment methodology provides 80 bits of encryption strength, which is no longer allowed in FIPS mode. (This is in addition to the the Approved 112 bits of encryption strength listed above.)
- Cryptographic Primitives Library has a non-approved algorithm implementation of AES GCM encryption.
- Cryptographic Primitives Library supports 2-Key Triple-DES Encryption, which is disallowed for usage altogether as of the end of 2015.
- Cryptographic Primitives Library also supports the following non FIPS 140-2 approved algorithms:
    o RSA encrypt/decrypt (disallowed in FIPS mode)
    o MD5 and HMAC-MD5 (allowed in TLS and EAP-TLS)
    o RC2, RC4, MD2, MD4 (disallowed in FIPS mode)
    o DES in ECB, CBC, CFB8 and CFB64 modes (disallowed in FIPS mode)
    o Legacy CAPI KDF (proprietary; disallowed in FIPS mode)
    o IEEE 1619-2007 XTS-AES (disallowed in FIPS mode)

---

[4] This cryptographic module supports the TLS, IKEv1, and IKEv2 protocols with SP 800-135 rev 1 KDF primitives, however, the protocols have not been reviewed or tested by the NIST CAVP and CMVP.

- ECDH with Curve25519 (allowed in FIPS mode as per FIPS 140-2 IG D.8 scenario 4). This curve is assumed to provide 128 bits of security strength[5].

2.3   Cryptographic Bypass

Cryptographic bypass is not supported by Cryptographic Primitives Library.

2.4   Machine Configurations

Cryptographic Primitives Library as tested using the machine configurations listed in Section 1.3 - Validated Platforms.

3   Operational Environment

The operational environment for Cryptographic Primitives Library is Windows 10 OEs running on the hardware listed in Section 1.3 - Validated Platforms.

Because Cryptographic Primitives Library module is a DLL, each process requesting access is provided its own instance of the module. As such, each process has full access to all information and keys within the module. Note that no keys or other information are maintained upon detachment from the DLL, thus an instantiation of the module will only contain keys or information that the process has placed in the module. BCRYPTPRIMITIVES.DLL and NCRYPTSSLP.DLL rely on the operating environment to enforce isolation between processes.

4   Integrity Chain of Trust

The integrity of Cryptographic Primitives Library is checked by Code Integrity before it is loaded into memory. This integrity check is based on the verification of an RSA signature over the binary using a 2048-bit key (Cert. # 2193) and a SHA-256 hash (Cert. # 3347), and verifying that the signing certificate chains up to a known root authority.

5   Ports and Interfaces

5.1   Export Functions

The Cryptographic Primitives Library module implements a set of algorithm providers for the Cryptography Next Generation (CNG) framework in Windows. Each provider in this module represents a single cryptographic algorithm or a set of closely related cryptographic algorithms. These algorithm providers are invoked through the CNG algorithm primitive functions, which are sometimes collectively referred to as the BCrypt API. For a full list of these algorithm providers, see:

https://msdn.microsoft.com/en-us/library/aa375534.aspx

---

[5] See Bernstein, Daniel J., *Curve25519: new Diffie-Hellman speed records*, URL:
https://cr.yp.to/ecdh/curve25519-20060209.pdf

Cryptographic Primitives Library

The Cryptographic Primitives Library module exposes its cryptographic services to the operating system through a small set of exported functions. These functions are used by the CNG framework to retrieve references to the different algorithm providers, in order to route BCrypt API calls appropriately to Cryptographic Primitives Library. These functions return references to implementations of cryptographic functions that correspond directly to functions in the BCrypt API. For details, please see the CNG SDK for Windows 10 OEs, available at:

https://msdn.microsoft.com/en-us/library/windows/desktop/aa376210.aspx
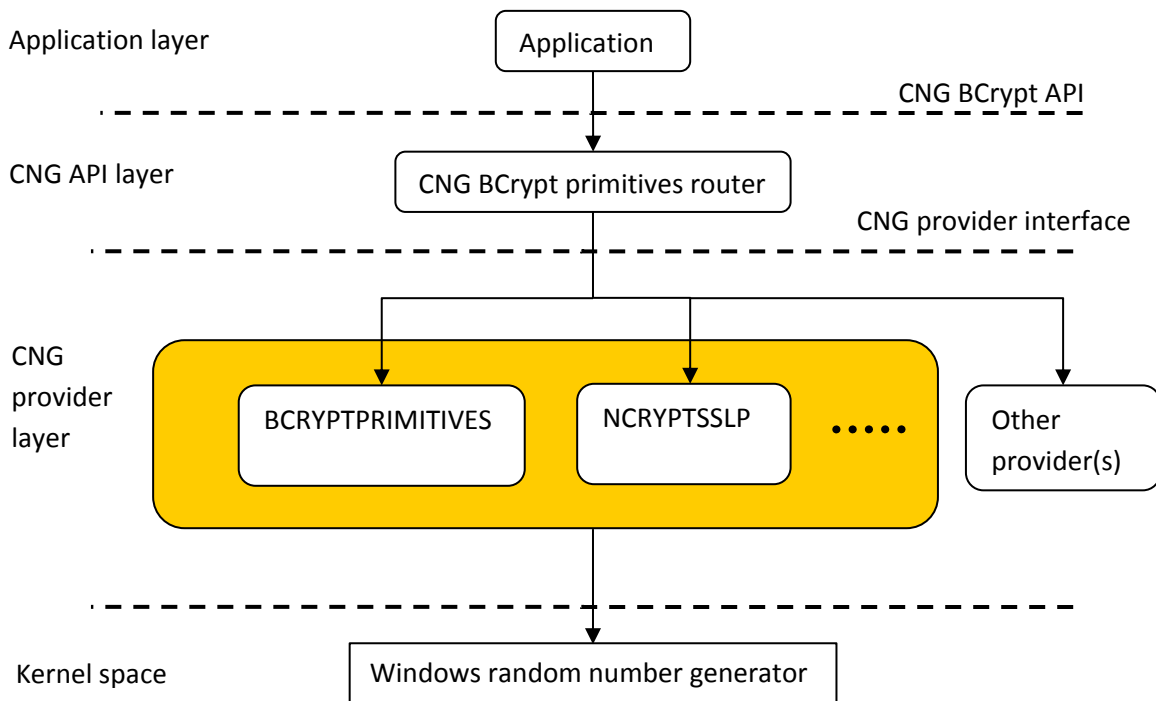
Figure 2 Relationships between Cryptographic Primitives Library and other components – cryptographic boundary highlighted in gold.

The following functions are exported by Cryptographic Primitives Library:

- GetAsymmetricEncryptionInterface
- GetCipherInterface
- GetHashInterface
- GetKeyDerivationInterface
- GetRngInterface
- GetSecretAgreementInterface
- GetSignatureInterface
- ProcessPrng

- ProcessPrngGuid

5.2 CNG Primitive Functions

The following list contains the CNG functions which can be used by callers to access the cryptographic services in Cryptographic Primitives Library.

- BCryptCloseAlgorithmProvider
- BCryptCreateHash
- BCryptCreateMultiHash
- BCryptDecrypt
- BCryptDeriveKey
- BCryptDeriveKeyPBKDF2
- BCryptDestroyHash
- BCryptDestroyKey
- BCryptDestroySecret
- BCryptDuplicateHash
- BCryptDuplicateKey
- BCryptEncrypt
- BCryptExportKey
- BCryptFinalizeKeyPair
- BCryptFinishHash
- BCryptFreeBuffer
- BCryptGenerateKeyPair
- BCryptGenerateSymmetricKey
- BCryptGenRandom
- BCryptGetProperty
- BCryptHash
- BCryptHashData
- BCryptImportKey
- BCryptImportKeyPair
- BCryptKeyDerivation
- BCryptOpenAlgorithmProvider
- BCryptProcessMultiOperations
- BCryptSecretAgreement
- BCryptSetProperty
- BCryptSignHash
- BCryptVerifySignature

All of these functions are used in the approved mode. Furthermore, these are the only approved functions that this module can perform.

Cryptographic Primitives Library has additional export functions described in Section 5.7 Non-Security Relevant Configuration Interfaces.

5.3   Control Input Interface

The Control Input Interface for Cryptographic Primitives Library consists of the CNG primitive functions listed in Section ⍰. Options for control operations are passed as input parameters to the CNG primitive functions.

5.4   Status Output Interface

The Status Output Interface for Cryptographic Primitives Library also consists of the CNG primitive functions listed in Section ⍰. For each function, the status information is returned to the caller as the return value from the function.

5.5   Data Output Interface

The Data Output Interface for Cryptographic Primitives Library also consists of the CNG primitive functions listed in Section ⍰.

5.6   Data Input Interface

The Data Input Interface for Cryptographic Primitives Library also consists of the CNG primitive functions listed in Section ⍰. Data and options are passed to the interface as input parameters to the CNG primitive functions. Data Input is kept separate from Control Input by passing Data Input in separate parameters from Control Input.

5.7   Non-Security Relevant Configuration Interfaces

These are not cryptographic functions. They are used to configure cryptographic providers on the system, and are provided for informational purposes. Please see https://msdn.microsoft.com for details.

Table 1

Function Name	Description
BCryptEnumAlgorithms	Enumerates the algorithms for a given set of operations.
BCryptEnumProviders	Returns a list of CNG providers for a given algorithm.
BCryptRegisterConfigChangeNotify	This is deprecated beginning with Windows 10.
BCryptResolveProviders	Resolves queries against the set of providers currently registered on the local system and the configuration information specified in the machine and domain configuration tables, returning an ordered list of references to one or more providers matching the specified criteria.
BCryptAddContextFunctionProvider	Adds a cryptographic function provider to the list of providers that are supported by an existing CNG context.
BCryptRegisterProvider	Registers a CNG provider.
BCryptUnregisterProvider	Unregisters a CNG provider.
BCryptUnregisterConfigChangeNotify	Removes a CNG configuration change event handler. This API differs slightly between User-Mode and Kernel-Mode.
BCryptGetFipsAlgorithmMode	Determines whether Cryptographic Primitives Library is operating in FIPS mode. Some applications use the value

	returned by this API to alter their own behavior, such as blocking the use of some SSL versions.

6 Specification of Roles

Cryptographic Primitives Library provides User and Cryptographic Officer roles (as defined in FIPS 140-2). These roles share all the services implemented in the cryptographic module.

When an application requests the crypto module to generate keys for a user, the keys are generated, used, and deleted as requested by applications. There are no implicit keys associated with a user. Each user may have numerous keys, and each user's keys are separate from other users' keys.

6.1 Maintenance Roles

Maintenance roles are not supported.

6.2 Multiple Concurrent Interactive Operators

There is only one interactive operator in Single User Mode. When run in this configuration, multiple concurrent interactive operators are not supported. This configuration is set by disabling user switching and disabling remote desktop access.

1. Open the Group Policy Editor by right-clicking the Start menu and selecting Run. Type gpedit.msc in the Run box and click OK.
2. Go to Local Computer Policy >> Computer Configuration >> Administrative Templates >> System >> Logon.
3. On the right side of the window, open "Hide entry points for Fast User Switching" and set the Enabled option. Click the OK button.
4. Open System Properties by right-clicking the Start menu and selecting System. Click the Remote Settings link.
5. Select the "Don't allow remote connections to this computer" option. Click the OK button.

6.3 Operator Authentication

The module does not provide authentication. Roles are implicitly assumed based on the services that are executed.

7 Services

The following list contains all services available to an operator. All services are accessible to both the User and Crypto Officer roles. In addition, there is zeroization (see Section 9 Cryptographic Key Management).

7.1 Algorithm Providers and Properties

7.1.1 BCryptOpenAlgorithmProvider

```
NTSTATUS WINAPI BCryptOpenAlgorithmProvider(
        BCRYPT_ALG_HANDLE  *phAlgorithm,
        LPCWSTR pszAlgId,
        LPCWSTR pszImplementation,
        ULONG   dwFlags);
```

The BCryptOpenAlgorithmProvider() function has four parameters: algorithm handle output to the opened algorithm provider, desired algorithm ID input, an optional specific provider name input, and optional flags. This function loads and initializes a CNG provider for a given algorithm, and returns a handle to the opened algorithm provider on success. See https://msdn.microsoft.com for CNG providers. Unless the calling function specifies the name of the provider, the default provider is used. The default provider is the first provider listed for a given algorithm. The calling function must pass the BCRYPT_ALG_HANDLE_HMAC_FLAG flag in order to use an HMAC function with a hash algorithm.

7.1.2 BCryptCloseAlgorithmProvider

```
NTSTATUS WINAPI BCryptCloseAlgorithmProvider(
        BCRYPT_ALG_HANDLE  hAlgorithm,
        ULONG   dwFlags);
```

This function closes an algorithm provider handle opened by a call to BCryptOpenAlgorithmProvider() function.

7.1.3 BCryptSetProperty

```
NTSTATUS WINAPI BCryptSetProperty(
        BCRYPT_HANDLE  hObject,
        LPCWSTR pszProperty,
        PUCHAR  pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptSetProperty() function sets the value of a named property for a CNG object, e.g., a cryptographic key. The CNG object is referenced by a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

7.1.4 BCryptGetProperty

```
NTSTATUS WINAPI BCryptGetProperty(
        BCRYPT_HANDLE  hObject,
        LPCWSTR pszProperty,
        PUCHAR  pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptGetProperty() function retrieves the value of a named property for a CNG object, e.g., a cryptographic key. The CNG object is referenced by a handle, the property name is a NULL terminated string, and the value of the property is a length-specified byte string.

7.1.5   BCryptFreeBuffer

```
VOID WINAPI BCryptFreeBuffer(
        PVOID   pvBuffer);
```

Some of the CNG functions allocate memory on caller's behalf. The BCryptFreeBuffer() function frees memory that was allocated by such a CNG function.

7.2   Random Number Generation

7.2.1   BCryptGenRandom

```
NTSTATUS WINAPI BCryptGenRandom(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        PUCHAR   pbBuffer,
        ULONG   cbBuffer,
        ULONG   dwFlags);
```

The BCryptGenRandom() function fills a buffer with random bytes. BCRYPTPRIMITVES.DLL implements the following random number generation algorithm:

- BCRYPT_RNG_ALGORITHM. This is the AES-256 counter mode based random generator as defined in SP 800-90A.

During the function initialization, a seed is obtained from the output of an in-kernel random number generator. This DRBG, which exists beyond the cryptographic boundary, provides the necessary entropy for the user-level DRBGs available through this function. A description of the entropy collection is documented with the SystemPrng function in the Kernel Mode Cryptographic Primitives Library (cng.sys) security policy.

7.3   Key and Key-Pair Generation

7.3.1   BCryptGenerateSymmetricKey

```
NTSTATUS WINAPI BCryptGenerateSymmetricKey(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbSecret,
        ULONG   cbSecret,
        ULONG   dwFlags);
```

The BCryptGenerateSymmetricKey() function generates a symmetric key object for use with a symmetric encryption or key derivation algorithm from a supplied *cbSecret* bytes long key value provided in the *pbSecret* memory location. The calling application must specify a handle to the algorithm provider

opened with the BCryptOpenAlgorithmProvider() function. The algorithm specified when the provider was opened must support symmetric key encryption or key derivation.

7.3.2 BCryptGenerateKeyPair

```
NTSTATUS WINAPI BCryptGenerateKeyPair(
        BCRYPT_ALG_HANDLE   hAlgorithm,
        BCRYPT_KEY_HANDLE   *phKey,
        ULONG   dwLength,
        ULONG   dwFlags);
```

The BCryptGenerateKeyPair() function creates a public/private key pair object without any cryptographic keys in it. After creating such an empty key pair object using this function, call the BCryptSetProperty() function to set its properties. The key pair can be used only after BCryptFinalizeKeyPair() function is called.

Note: for when generating a key pair with "BCRYPT_DSA_ALGORITHM" If the key length is 1024 bits, then a process conformant with FIPS 186-2 DSA will be used to generate the key pair and perform subsequent DSA operations. If the key length is 2048 or 3072 bits, then a process conformant with FIPS 186-3 DSA is used to generate the key pair and perform subsequent DSA operations.

7.3.3 BCryptFinalizeKeyPair

```
NTSTATUS WINAPI BCryptFinalizeKeyPair(
        BCRYPT_KEY_HANDLE   hKey,
        ULONG   dwFlags);
```

The BCryptFinalizeKeyPair() function completes a public/private key pair import or generation. The key pair cannot be used until this function has been called. After this function has been called, the BCryptSetProperty() function can no longer be used for this key pair.

7.3.4 BCryptDuplicateKey

```
NTSTATUS WINAPI BCryptDuplicateKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   *phNewKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        ULONG   dwFlags);
```

The BCryptDuplicateKey() function creates a duplicate of a symmetric key object.

7.3.5 BCryptDestroyKey

```
NTSTATUS WINAPI BCryptDestroyKey(
        BCRYPT_KEY_HANDLE   hKey);
```

The BCryptDestroyKey() function destroys a key.

7.4   Key Entry and Output

7.4.1   BCryptImportKey

```
NTSTATUS WINAPI BCryptImportKey(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbKeyObject,
        ULONG   cbKeyObject,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```

The BCryptImportKey() function imports a symmetric key from a key blob.

hAlgorithm [in] is the handle of the algorithm provider to import the key. This handle is obtained by calling the BCryptOpenAlgorithmProvider function.

hImportKey [in, out] is not currently used and should be NULL.

pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB that is contained in the *pbInput* buffer. *pszBlobType* can be one of BCRYPT_AES_WRAP_KEY_BLOB, BCRYPT_KEY_DATA_BLOB and BCRYPT_OPAQUE_KEY_BLOB.

phKey [out] is a pointer to a BCRYPT_KEY_HANDLE that receives the handle of the imported key that is used in subsequent functions that require a key, such as BCryptEncrypt. This handle must be released when it is no longer needed by passing it to the BCryptDestroyKey function.

pbKeyObject [out] is a pointer to a buffer that receives the imported key object. The *cbKeyObject* parameter contains the size of this buffer. The required size of this buffer can be obtained by calling the BCryptGetProperty function to get the BCRYPT_OBJECT_LENGTH property. This will provide the size of the key object for the specified algorithm. This memory can only be freed after the *phKey* key handle is destroyed.

cbKeyObject [in] is the size, in bytes, of the pbKeyObject buffer.

pbInput [in] is the address of a buffer that contains the key BLOB to import.

The *cbInput* parameter contains the size of this buffer.

The *pszBlobType* parameter specifies the type of key BLOB this buffer contains.

cbInput [in] is the size, in bytes, of the pbInput buffer.

dwFlags [in] is a set of flags that modify the behavior of this function. No flags are currently defined, so this parameter should be zero.

7.4.2   BCryptImportKeyPair
```
NTSTATUS WINAPI BCryptImportKeyPair(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_KEY_HANDLE hImportKey,
        LPCWSTR pszBlobType,
        BCRYPT_KEY_HANDLE *phKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        ULONG   dwFlags);
```
The BCryptImportKeyPair() function is used to import a public/private key pair from a key blob.

hAlgorithm [in] is the handle of the algorithm provider to import the key. This handle is obtained by calling the BCryptOpenAlgorithmProvider function.

hImportKey [in, out] is not currently used and should be NULL.

pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB that is contained in the pbInput buffer. This can be one of the following values: BCRYPT_DH_PRIVATE_BLOB, BCRYPT_DH_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB, BCRYPT_DSA_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB_V2, BCRYPT_DSA_PUBLIC_BLOB_V2, BCRYPT_ECCPRIVATE_BLOB, BCRYPT_ECCPUBLIC_BLOB, BCRYPT_PUBLIC_KEY_BLOB, BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAPRIVATE_BLOB, BCRYPT_RSAPUBLIC_BLOB, LEGACY_DH_PUBLIC_BLOB, LEGACY_DH_PRIVATE_BLOB, LEGACY_DSA_PRIVATE_BLOB, LEGACY_DSA_PUBLIC_BLOB, LEGACY_DSA_V2_PRIVATE_BLOB, LEGACY_RSAPRIVATE_BLOB, LEGACY_RSAPUBLIC_BLOB.

Note:

BCRYPT_DSA_PRIVATE_BLOB and BCRYPT_DSA_PUBLIC_BLOB are used for 1024-bit DSA key lengths.

BCRYPT_DSA_PRIVATE_BLOB_V2, BCRYPT_DSA_PUBLIC_BLOB_V2 are used for 2048-bit and 3072-bit DSA key lengths.

phKey [out] is a pointer to a BCRYPT_KEY_HANDLE that receives the handle of the imported key. This handle is used in subsequent functions that require a key, such as BCryptSignHash. This handle must be released when it is no longer needed by passing it to the BCryptDestroyKey function.

pbInput [in] is the address of a buffer that contains the key BLOB to import. The cbInput parameter contains the size of this buffer. The pszBlobType parameter specifies the type of key BLOB this buffer contains.

cbInput [in] contains the size, in bytes, of the pbInput buffer.

dwFlags [in] is a set of flags that modify the behavior of this function. This can be zero or the following value: BCRYPT_NO_KEY_VALIDATION.

7.4.3 BCryptExportKey

```
NTSTATUS WINAPI BCryptExportKey(
        BCRYPT_KEY_HANDLE   hKey,
        BCRYPT_KEY_HANDLE   hExportKey,
        LPCWSTR pszBlobType,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptExportKey() function exports a key to a memory blob that can be persisted for later use.

hKey [in] is the handle of the key to export.

hExportKey [in, out] is not currently used and should be set to NULL.

pszBlobType [in] is a null-terminated Unicode string that contains an identifier that specifies the type of BLOB to export. This can be one of the following values: BCRYPT_AES_WRAP_KEY_BLOB, BCRYPT_DH_PRIVATE_BLOB, BCRYPT_DH_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB, BCRYPT_DSA_PUBLIC_BLOB, BCRYPT_DSA_PRIVATE_BLOB_V2, BCRYPT_DSA_PUBLIC_BLOB_V2, BCRYPT_ECCPRIVATE_BLOB, BCRYPT_ECCPUBLIC_BLOB, BCRYPT_KEY_DATA_BLOB, BCRYPT_OPAQUE_KEY_BLOB, BCRYPT_PUBLIC_KEY_BLOB, BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAPRIVATE_BLOB, BCRYPT_RSAPUBLIC_BLOB, LEGACY_DH_PRIVATE_BLOB, LEGACY_DH_PUBLIC_BLOB, LEGACY_DSA_PRIVATE_BLOB, LEGACY_DSA_PUBLIC_BLOB, LEGACY_DSA_V2_PRIVATE_BLOB, LEGACY_RSAPRIVATE_BLOB, LEGACY_RSAPUBLIC_BLOB.

Note:

BCRYPT_DSA_PRIVATE_BLOB and BCRYPT_DSA_PUBLIC_BLOB are used for 1024-bit DSA key lengths.

BCRYPT_DSA_PRIVATE_BLOB_V2, BCRYPT_DSA_PUBLIC_BLOB_V2 are used for 2048-bit and 3072-bit DSA key lengths.

pbOutput is the address of a buffer that receives the key BLOB. The cbOutput parameter contains the size of this buffer. If this parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by the pcbResult parameter.

cbOutput [in] contains the size, in bytes, of the pbOutput buffer.

pcbResult [out] is a pointer to a ULONG that receives the number of bytes that were copied to the pbOutput buffer. If the pbOutput parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by this parameter.

dwFlags [in] is a set of flags that modify the behavior of this function. No flags are defined for this function.

7.5   Encryption and Decryption
7.5.1   BCryptEncrypt

```
NTSTATUS WINAPI BCryptEncrypt(
        BCRYPT_KEY_HANDLE hKey,
        PUCHAR   pbInput,
        ULONG   cbInput,
        VOID    *pPaddingInfo,
        PUCHAR   pbIV,
        ULONG   cbIV,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptEncrypt() function encrypts a block of data of given length.

hKey [in, out] is the handle of the key to use to encrypt the data. This handle is obtained from one of the key creation functions, such as BCryptGenerateSymmetricKey, BCryptGenerateKeyPair, or BCryptImportKey.

pbInput [in] is the address of a buffer that contains the plaintext to be encrypted. The cbInput parameter contains the size of the plaintext to encrypt. For more information, see Remarks.

cbInput [in] is the number of bytes in the pbInput buffer to encrypt.

pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and authenticated encryption modes (i.e. AES-CCM and AES-GCM). It must be NULL otherwise.

pbIV [in, out, optional] is the address of a buffer that contains the initialization vector (IV)[6] to use during encryption. The cbIV parameter contains the size of this buffer. This function will modify the contents of this buffer. If you need to reuse the IV later, make sure you make a copy of this buffer before calling this function. This parameter is optional and can be NULL if no IV is used. The required size of the IV can be

---

[6] The IV being passed into the BCryptEncrypt() API needs to be generated by a validated cryptographic module or an approved DRBG.

obtained by calling the BCryptGetProperty function to get the BCRYPT_BLOCK_LENGTH property. This will provide the size of a block for the algorithm, which is also the size of the IV.

cbIV [in] contains the size, in bytes, of the pbIV buffer.

pbOutput [out, optional] is the address of a buffer that will receive the ciphertext produced by this function. The cbOutput parameter contains the size of this buffer. For more information, see Remarks.

If this parameter is NULL, this function will calculate the size needed for the ciphertext and return the size in the location pointed to by the pcbResult parameter.

cbOutput [in] contains the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.

pcbResult [out] is a pointer to a ULONG variable that receives the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the ciphertext.

dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this can be zero or the following value: BCRYPT_BLOCK_PADDING. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_NONE, BCRYPT_PAD_OAEP, BCRYPT_PAD_PKCS1.

7.5.2   BCryptDecrypt
```
NTSTATUS WINAPI BCryptDecrypt(
        BCRYPT_KEY_HANDLE  hKey,
        PUCHAR  pbInput,
        ULONG  cbInput,
        VOID  *pPaddingInfo,
        PUCHAR  pbIV,
        ULONG  cbIV,
        PUCHAR  pbOutput,
        ULONG  cbOutput,
        ULONG  *pcbResult,
        ULONG  dwFlags);
```
The BCryptDecrypt() function decrypts a block of data of given length.

hKey [in, out] is the handle of the key to use to decrypt the data. This handle is obtained from one of the key creation functions, such as BCryptGenerateSymmetricKey, BCryptGenerateKeyPair, or BCryptImportKey.

pbInput [in] is the address of a buffer that contains the ciphertext to be decrypted. The cbInput parameter contains the size of the ciphertext to decrypt. For more information, see Remarks.

cbInput [in] is the number of bytes in the pbInput buffer to decrypt.

Page 24 of 48

pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and authenticated encryption modes (i.e. AES-CCM and AES-GCM). It must be NULL otherwise.

pbIV [in, out, optional] is the address of a buffer that contains the initialization vector (IV) to use during decryption. The cbIV parameter contains the size of this buffer. This function will modify the contents of this buffer. If you need to reuse the IV later, make sure you make a copy of this buffer before calling this function. This parameter is optional and can be NULL if no IV is used. The required size of the IV can be obtained by calling the BCryptGetProperty function to get the BCRYPT_BLOCK_LENGTH property. This will provide the size of a block for the algorithm, which is also the size of the IV.

cbIV [in] contains the size, in bytes, of the pbIV buffer.

pbOutput [out, optional] is the address of a buffer to receive the plaintext produced by this function. The cbOutput parameter contains the size of this buffer. For more information, see Remarks.

If this parameter is NULL, this function will calculate the size required for the plaintext and return the size in the location pointed to by the pcbResult parameter.

cbOutput [in] is the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.

pcbResult [out] is a pointer to a ULONG variable to receive the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the plaintext.

dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this can be zero or the following value: BCRYPT_BLOCK_PADDING. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_NONE, BCRYPT_PAD_OAEP, BCRYPT_PAD_PKCS1.

7.6 Hashing and Message Authentication

7.6.1 BCryptCreateHash

```
NTSTATUS WINAPI BCryptCreateHash(
        BCRYPT_ALG_HANDLE  hAlgorithm,
        BCRYPT_HASH_HANDLE *phHash,
        PUCHAR  pbHashObject,
        ULONG  cbHashObject,
        PUCHAR  pbSecret,
        ULONG  cbSecret,
        ULONG  dwFlags);
```

The BCryptCreateHash() function creates a hash object with an optional key. The optional key is used for HMAC, AES GMAC and AES CMAC.

hAlgorithm [in, out] is the handle of an algorithm provider created by using the BCryptOpenAlgorithmProvider function. The algorithm that was specified when the provider was created must support the hash interface.

phHash [out] is a pointer to a BCRYPT_HASH_HANDLE value that receives a handle that represents the hash object. This handle is used in subsequent hashing functions, such as the BCryptHashData function. When you have finished using this handle, release it by passing it to the BCryptDestroyHash function.

pbHashObject [out] is a pointer to a buffer that receives the hash object. The cbHashObject parameter contains the size of this buffer. The required size of this buffer can be obtained by calling the BCryptGetProperty function to get the BCRYPT_OBJECT_LENGTH property. This will provide the size of the hash object for the specified algorithm. This memory can only be freed after the hash handle is destroyed.

cbHashObject [in] contains the size, in bytes, of the pbHashObject buffer.

pbSecret [in, optional] is a pointer to a buffer that contains the key to use for the hash. The cbSecret parameter contains the size of this buffer. If no key should be used with the hash, set this parameter to NULL. This key only applies to the HMAC, AES GMAC and AES CMAC algorithms.

cbSecret [in, optional] contains the size, in bytes, of the pbSecret buffer. If no key should be used with the hash, set this parameter to zero.

dwFlags [in] is not currently used and must be zero.

7.6.2 BCryptHashData

```
NTSTATUS WINAPI BCryptHashData(
        BCRYPT_HASH_HANDLE  hHash,
        PUCHAR  pbInput,
        ULONG  cbInput,
        ULONG  dwFlags);
```

The BCryptHashData() function performs a one way hash on a data buffer. Call the BCryptFinishHash() function to finalize the hashing operation to get the hash result.

7.6.3   BCryptDuplicateHash

```
NTSTATUS WINAPI BCryptDuplicateHash(
        BCRYPT_HASH_HANDLE  hHash,
        BCRYPT_HASH_HANDLE  *phNewHash,
        PUCHAR   pbHashObject,
        ULONG   cbHashObject,
        ULONG   dwFlags);
```

The BCryptDuplicateHash()function duplicates an existing hash object. The duplicate hash object contains all state and data that was hashed to the point of duplication.

7.6.4   BCryptFinishHash

```
NTSTATUS WINAPI BCryptFinishHash(
        BCRYPT_HASH_HANDLE hHash,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   dwFlags);
```

The BCryptFinishHash() function retrieves the hash value for the data accumulated from prior calls to BCryptHashData() function.

7.6.5   BCryptDestroyHash

```
NTSTATUS WINAPI BCryptDestroyHash(
        BCRYPT_HASH_HANDLE  hHash);
```

The BCryptDestroyHash() function destroys a hash object.

7.6.6   BCryptHash

```
NTSTATUS WINAPI BCryptHash(
        BCRYPT_ALG_HANDLE hAlgorithm,
        PUCHAR pbSecret,
        ULONG cbSecret,
        PUCHAR pbInput,
        ULONG cbInput,
        PUCHAR pbOutput,
        ULONG  cbOutput);
```

The function BCryptHash() performs a single hash computation. This is a convenience function that wraps calls to the BCryptCreateHash(), BCryptHashData(), BCryptFinishHash(), and BCryptDestroyHash() functions.

7.6.7   BCryptCreateMultiHash

```
NTSTATUS WINAPI BCryptCreateMultiHash(
        BCRYPT_ALG_HANDLE hAlgorithm,
        BCRYPT_HASH_HANDLE *phHash,
        ULONG nHashes,
        PUCHAR pbHashObject,
        ULONG cbHashObject,
        PUCHAR pbSecret,
        ULONG cbSecret,
        ULONG dwFlags);
```

BCryptCreateMultiHash() is a function that creates a new MultiHash object that is used in parallel hashing to improve performance. The MultiHash object is equivalent to an array of normal (reusable) hash objects.

hAlgorithm [in out] is the handle of an algorithm provider. See the BCryptCreateHash() description above for details.

phHash [out] is a pointer to the hHash handle to be returned by this function. This hHash handle can be freed using the BCryptDestroyHash() function. Use of this particular hHash in BCryptDuplicateHash() is currently not supported by the default algorithm provider.

nHashes [in] nHashes is the number of entries in the array. nHashes must be at least 1. The default provider implements an upper bound of 64 hash states.

pbHashObject [out, optional] is a pointer to a buffer that receives the hash object. See the BCryptCreateHash() description above for details.

cbHashObject [in] contains the size, in bytes, of the pbHashObject buffer.

pbSecret [in, optional] specifies the key (if any) for all of the hash computations that will be done using this MultiHash object. The secret is used for all of the array entries.

cbSecret [in] is the size in bytes of the pbSecret buffer.

dwFlags [in]] is not currently used and must be zero.

7.6.8 BCryptProcessMultiOperations

```
NTSTATUS WINAPI BCryptProcessMultiOperations(
        BCRYPT_HANDLE  hObject,
        BCRYPT_MULTI_OPERATION_TYPE operationType,
        PVOID pOperations,
        ULONG cbOperations,
        ULONG dwFlags );
```

The BCryptProcessMultiOperations() function is used to perform multiple operations on a single multi-object handle such as a MultiHash object handle. If any of the operations fail, then the function will return an error.

hObject [in out] is the hObject handle referring to the multi-object.

operationType [in] specifies the type of operation structure being passed in the pOperations/cbOperations buffer.

pOperations [in] is a pointer that must point to a valid array of the specified type; e.g. alignment requirements must be met.

cbOperations [in] must be nonzero and a multiple of the sizeof of the specified type.

dwFlags [in] is not currently used and must be zero.

Each element of the operations array specifies an operation to be performed on/with the hObject.

For hash operations, there are two operation types:

- Hash data
- Finalize hash

These correspond directly to BCryptHashData() and BCryptFinishHash(). Each operation specifies an index of the hash object inside the hObject MultiHash object that this operation applies to. Operations are executed in any order or even in parallel, with the sole restriction that the set of operations that specify the same index are all executed in-order.

7.7   Signing and Verification
7.7.1   BCryptSignHash

```
NTSTATUS WINAPI BCryptSignHash(
        BCRYPT_KEY_HANDLE   hKey,
        VOID   *pPaddingInfo,
        PUCHAR   pbInput,
        ULONG   cbInput,
        PUCHAR   pbOutput,
        ULONG   cbOutput,
        ULONG   *pcbResult,
        ULONG   dwFlags);
```

The BCryptSignHash() function creates a signature of a hash value.

hKey [in] is the handle of the key to use to sign the hash.

pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the dwFlags parameter. This parameter is only used with asymmetric keys and must be NULL otherwise.

pbInput [in] is a pointer to a buffer that contains the hash value to sign. The cbInput parameter contains the size of this buffer.

cbInput [in] is the number of bytes in the pbInput buffer to sign.

pbOutput [out] is the address of a buffer to receive the signature produced by this function. The cbOutput parameter contains the size of this buffer. If this parameter is NULL, this function will calculate the size required for the signature and return the size in the location pointed to by the pcbResult parameter.

cbOutput [in] is the size, in bytes, of the pbOutput buffer. This parameter is ignored if the pbOutput parameter is NULL.

pcbResult [out] is a pointer to a ULONG variable that receives the number of bytes copied to the pbOutput buffer. If pbOutput is NULL, this receives the size, in bytes, required for the signature.

dwFlags [in] is a set of flags that modify the behavior of this function. The allowed set of flags depends on the type of key specified by the hKey parameter. If the key is a symmetric key, this parameter is not used and should be set to zero. If the key is an asymmetric key, this can be one of the following values: BCRYPT_PAD_PKCS1, BCRYPT_PAD_PSS.

Note: this function accepts SHA-1 hashes, which according to NIST SP 800-131A is *disallowed* for digital signature generation. SHA-1 is currently *legacy-use* for digital signature verification.

7.7.2 BCryptVerifySignature

```
NTSTATUS WINAPI BCryptVerifySignature(
        BCRYPT_KEY_HANDLE   hKey,
        VOID    *pPaddingInfo,
        PUCHAR   pbHash,
        ULONG   cbHash,
        PUCHAR   pbSignature,
        ULONG   cbSignature,
        ULONG   dwFlags);
```

The BCryptVerifySignature() function verifies that the specified signature matches the specified hash.

hKey [in] is the handle of the key to use to decrypt the signature. This must be an identical key or the public key portion of the key pair used to sign the data with the BCryptSignHash function.

pPaddingInfo [in, optional] is a pointer to a structure that contains padding information. The actual type of structure this parameter points to depends on the value of the *dwFlags* parameter. This parameter is only used with asymmetric keys and must be NULL otherwise.

pbHash [in] is the address of a buffer that contains the hash of the data. The cbHash parameter contains the size of this buffer.

cbHash [in] is the size, in bytes, of the pbHash buffer.

pbSignature [in] is the address of a buffer that contains the signed hash of the data. The BCryptSignHash function is used to create the signature. The *cbSignature* parameter contains the size of this buffer.

cbSignature [in] is the size, in bytes, of the pbSignature buffer. The BCryptSignHash function is used to create the signature.

Note: this function accepts SHA-1 hashes, which according to NIST SP 800-131A is *disallowed* for digital signature generation. SHA-1 is currently *legacy-use* for digital signature verification.

7.8   Secret Agreement and Key Derivation

7.8.1   BCryptSecretAgreement

```
NTSTATUS WINAPI BCryptSecretAgreement(
        BCRYPT_KEY_HANDLE      hPrivKey,
        BCRYPT_KEY_HANDLE      hPubKey,
        BCRYPT_SECRET_HANDLE   *phAgreedSecret,
        ULONG               dwFlags);
```

The BCryptSecretAgreement() function creates a secret agreement value from a private and a public key. This function is used with Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH) algorithms.

hPrivKey [in] The handle of the private key to use to create the secret agreement value.

hPubKey [in] The handle of the public key to use to create the secret agreement value.

phSecret [out] A pointer to a BCRYPT_SECRET_HANDLE that receives a handle that represents the secret agreement value. This handle must be released by passing it to the BCryptDestroySecret function when it is no longer needed.

dwFlags [in] A set of flags that modify the behavior of this function. This must be zero.

7.8.2   BCryptDeriveKey

```
NTSTATUS WINAPI BCryptDeriveKey(
        BCRYPT_SECRET_HANDLE hSharedSecret,
        LPCWSTR            pwszKDF,
        BCryptBufferDesc    *pParameterList,
        PUCHAR pbDerivedKey,
        ULONG            cbDerivedKey,
        ULONG            *pcbResult,
        ULONG            dwFlags);
```

The BCryptDeriveKey() function derives a key from a secret agreement value.

hSharedSecret [in, optional] is the secret agreement handle to create the key from. This handle is obtained from the BCryptSecretAgreement function.

pwszKDF [in] is a pointer to a null-terminated Unicode string that contains an object identifier (OID) that identifies the key derivation function (KDF) to use to derive the key. This can be one of the following strings: BCRYPT_KDF_HASH (parameters in pParameterList: KDF_HASH_ALGORITHM, KDF_SECRET_PREPEND, KDF_SECRET_APPEND), BCRYPT_KDF_HMAC (parameters in pParameterList: KDF_HASH_ALGORITHM, KDF_HMAC_KEY, KDF_SECRET_PREPEND, KDF_SECRET_APPEND), BCRYPT_KDF_TLS_PRF (parameters in pParameterList: KDF_TLS_PRF_LABEL, KDF_TLS_PRF_SEED), BCRYPT_KDF_SP80056A_CONCAT (parameters in pParameterList: KDF_ALGORITHMID, KDF_PARTYUINFO, KDF_PARTYVINFO, KDF_SUPPPUBINFO, KDF_SUPPPRIVINFO).

pParameterList [in, optional] is the address of a BCryptBufferDesc structure that contains the KDF parameters. This parameter is optional and can be NULL if it is not needed.

Note: When supporting a key agreement scheme that requires a nonce, BCryptDeriveKey uses whichever nonce is supplied by the caller in the BCryptBufferDesc. Examples of the nonce types are found in Section 5.4 of http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

When using a nonce, a random nonce **should** be used. And then if the random nonce is used, the entropy (amount of randomness) of the nonce and the security strength of the DRBG has to be at least one half of the minimum required bit length of the subgroup order.

For example:

for KAS FFC, entropy of nonce must be 112 bits for FB, 128 bits for FC).

for KAS ECC, entropy of the nonce must be 128 bit for EC, 182 for ED, 256 for EF.

pbDerivedKey [out, optional] is the address of a buffer that receives the key. The cbDerivedKey parameter contains the size of this buffer. If this parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by the pcbResult parameter.

cbDerivedKey [in] contains the size, in bytes, of the pbDerivedKey buffer.

pcbResult [out] is a pointer to a ULONG that receives the number of bytes that were copied to the pbDerivedKey buffer. If the pbDerivedKey parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by this parameter.

dwFlags [in] is a set of flags that modify the behavior of this function. This can be zero or KDF_USE_SECRET_AS_HMAC_KEY_FLAG. The KDF_USE_SECRET_AS_HMAC_KEY_FLAG value must only be used when pwszKDF is equal to BCRYPT_KDF_HMAC. It indicates that the secret will also be used as the HMAC key. If this flag is used, the KDF_HMAC_KEY parameter must not be specified in pParameterList.

7.8.3 BCryptDestroySecret
```
NTSTATUS WINAPI BCryptDestroySecret(
        BCRYPT_SECRET_HANDLE   hSecret);
```
The BCryptDestroySecret() function destroys a secret agreement handle that was created by using the BCryptSecretAgreement() function.

7.8.4 BCryptKeyDerivation

```
NTSTATUS WINAPI BCryptKeyDerivation(
        _In_      BCRYPT_KEY_HANDLE hKey,
        _In_opt_  BCryptBufferDesc   *pParameterList,
        _Out_writes_bytes_to_(cbDerivedKey, *pcbResult) PUCHAR pbDerivedKey,
        _In_      ULONG          cbDerivedKey,
        _Out_     ULONG           *pcbResult,
        _In_      ULONG          dwFlags);
```

The BCryptKeyDerivation() function executes a Key Derivation Function (KDF) on a key generated with BCryptGenerateSymmetricKey() function. It differs from the BCryptDeriveKey() function in that it does not require a secret agreement step to create a shared secret.

hKey [in] is a handle to a key created with the BCryptGenerateSymmetricKey function.

pParameterList [in] is the algorithm-specific parameter list for the selected KDF.

pbDerivedKey [out] is the address of a buffer that receives the key. The cbDerivedKey parameter contains the size of this buffer.

cbDerivedKey [in] contains the size, in bytes, of the pbDerivedKey buffer.

pcbResult [out] is a pointer to a ULONG that receives the number of bytes that were copied to the pbDerivedKey buffer. If the pbDerivedKey parameter is NULL, this function will place the required size, in bytes, in the ULONG pointed to by this parameter.

dwFlags [in] is a set of flags that modify the behavior of this function. This must be zero.

7.8.5 BCryptDeriveKeyPBKDF2

```
NTSTATUS WINAPI BCryptDeriveKeyPBKDF2(
        BCRYPT_ALG_HANDLE hPrf,
        PUCHAR pbPassword,
        ULONG cbPassword,
        PUCHAR pbSalt,
        ULONG cbSalt,
        ULONGLONG cIterations,
        PUCHAR pbDerivedKey,
        ULONG cbDerivedKey,
        ULONG dwFlags);
```

The BCryptDeriveKeyPBKDF2() function derives a key from a hash value by using the password based key derivation function as defined by SP 800-132 PBKDF and IETF RFC 2898 (specified as PBKDF2).

hPrf [in] is the handle of an algorithm provider that provides the pseudo-random function.

pbPassword [in, optional] is a pointer to a buffer that contains the password parameter for the PBKDF2 key derivation algorithm.

cbPassword [in] is the length, in bytes, of the data in the buffer pointed to by the *pbPassword* parameter.

pbSalt [in, optional] is a pointer to a buffer that contains the salt argument for the PBKDF2 key derivation algorithm.

cbSalt [in] is the length, in bytes, of the salt argument pointed to by the *pbSalt* parameter.

cIterations [in] is the iteration count for the PBKDF2 key derivation algorithm.

pbDerivedKey [out] is a pointer to a buffer that receives the derived key.

cbDerivedKey [in] is the length, in bytes, of the derived key returned in the buffer pointed to by the *pbDerivedKey* parameter.

dwFlags [in] This parameter is reserved and must be set to zero.

7.9 Deprecation

7.9.1 Bit Strengths of DH and ECDH
Through the year 2010, implementations of DH and ECDH were allowed to have an acceptable bit strength of at least 80 bits of security (for DH at least 1024 bits and for ECDH at least 160 bits). From 2011 through 2013, 80 bits of security strength was considered deprecated, and was disallowed starting January 1, 2014. As of that date, only security strength of at least 112 bits is acceptable. ECDH uses curve sizes of at least 256 bits (that means it has at least 128 bits of security strength), so that is acceptable. However, DH has a range of 1024 to 4096 and that changed to 2048 to 4096 after 2013.

7.9.2 SHA-1
From 2011 through 2013, SHA-1 could be used in a deprecated mode for use in digital signature generation. As of Jan. 1, 2014, SHA-1 is no longer allowed for digital signature generation, and it is allowed for legacy use only for digital signature verification.

7.10 Show Status Services
The User and Cryptographic Officer roles have the same Show Status functionality, which is, for each function, the status information is returned to the caller as the return value from the function.

7.11 Self-Test Services
The User and Cryptographic Officer roles have the same Self-Test functionality, which is described in Section 10 Self-Tests.

7.12 Service Inputs / Outputs

The User and Cryptographic Officer roles have service inputs and outputs as specified in Section 5 Ports and Interfaces and as described above.

7.13 Mapping of Services, Algorithms, and Critical Security Parameters

The following table maps the services to their corresponding algorithms and critical security parameters (CSPs).

Table 2

Service	Algorithms	CSPs
Algorithm Providers and Properties	None	None
Random Number Generation	AES-256 CTR DRBG	AES-CTR DRBG Seed AES-CTR DRBG Entropy Input AES-CTR DRBG V AES-CTR DRBG Key
Key and Key-Pair Generation	RSA, DH, ECDH, ECDSA, RC2, RC4, DES, Triple-DES, AES, and HMAC (RC2, RC4, and DES cannot be used in FIPS mode.)	Symmetric Keys Asymmetric Public Keys Asymmetric Private Keys
Key Entry and Output	SP 800-38F AES Key Wrapping (128, 192, and 256)	Symmetric Keys Asymmetric Public Keys Asymmetric Private Keys
Encryption and Decryption	Triple-DES with 2 key (encryption disallowed) and 3 key in ECB, CBC, CFB8 and CFB64 modes; AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes; AES-128, AES-192, and AES-256 in CCM, CMAC, and GMAC modes; AES-128, AES-192, and AES-256 GCM decryption; XTS-AES XTS-128 and XTS-256; SP 800-56B RSADP mod 2048; (RC2, RC4, RSA, and DES, which cannot be used in FIPS mode) IEEE 1619-2007 XTS-AES (non-FIPS Approved algorithm)	Symmetric Keys Asymmetric Public Keys Asymmetric Private Keys
Hashing and Message Authentication	FIPS 180-4 SHS SHA-1, SHA-256, SHA-384, and SHA-512; FIPS 180-4 SHA-1, SHA-256, SHA-384, SHA-512 HMAC; AES-128, AES-192, and AES-256 in CCM, CMAC, and GMAC;	Symmetric Keys (for HMAC, AES CCM, AES CMAC, and AES GMAC)

	MD5 and HMAC-MD5 (allowed in TLS and EAP-TLS); MD2 and MD4 (disallowed in FIPS mode)	
Signing and Verification	FIPS 186-4 RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signature generation and verification with 2048 and 3072 modulus; supporting SHA-1[7], SHA-256, SHA-384, and SHA-512 FIPS 186-4 ECDSA with the following NIST curves: P-256, P-384, P-521	Asymmetric RSA Public Keys Asymmetric RSA Private Keys Asymmetric ECDSA Public keys Asymmetric ECDSA Private keys
Secret Agreement and Key Derivation	KAS – SP 800-56A Diffie-Hellman Key Agreement; Finite Field Cryptography (FFC) KAS – SP 800-56A EC Diffie-Hellman Key Agreement SP 800-108 Key Derivation Function (KDF) CMAC-AES (128, 192, 256), HMAC (SHA1, SHA-256, SHA-384, SHA-512) SP 800-132 PBKDF Legacy CAPI KDF (cannot be used in FIPS mode)	DH Private and Public Values ECDH Private and Public Values
Show Status	None	None
Self-Tests	See Section **Error! Reference source not found.** Self-Tests for the list of algorithms	None
Zeroization	None	None

7.14 Mapping of Services, Export Functions, and Invocations

The following table maps the services to their corresponding export functions and invocations.

Table 3

Service	Export Functions	Invocations
Algorithm Providers and Properties	BCryptOpenAlgorithmProvider BCryptCloseAlgorithmProvider BCryptSetProperty BCryptGetProperty BCryptFreeBuffer	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Random Number Generation	BcryptGenRandom	The User / Cryptographic

[7] SHA-1 is only acceptable for signature verification.

		Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Key and Key-Pair Generation	BCryptGenerateSymmetricKey BCryptGenerateKeyPair BCryptFinalizeKeyPair BCryptDuplicateKey BCryptDestroyKey	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Key Entry and Output	BCryptImportKey BCryptImportKeyPair BCryptExportKey	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Encryption and Decryption	BCryptEncrypt BCryptDecrypt	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Hashing and Message Authentication	BCryptCreateHash BCryptHashData BCryptDuplicateHash BCryptFinishHash BCryptDestroyHash BCryptHash BCryptCreateMultiHash BCryptProcessMultiOperations	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Signing and Verification	BCryptSignHash BCryptVerifySignature	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.
Secret Agreement and Key Derivation	BCryptSecretAgreement BCryptDeriveKey BCryptDestroySecret BCryptKeyDerivation BCryptDeriveKeyPBKDF2	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of

		these exported functions is called.
Show Status	All Exported Functions	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed upon completion of an exported function.
Self-Tests	DLL Entry Point	This service is fully automatic. The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed upon startup of this module.
Zeroization	BCryptDestroyKey BCryptDestroySecret	The User / Cryptographic Officer does not take any actions to explicitly start this service. This service is executed whenever one of these exported functions is called.

7.15 Non-Approved Services

The following table lists other non-approved APIs exported from the crypto module.

Table 4

Function Name	Description
BCryptDeriveKeyCapi	Derives a key from a hash value. This function is provided as a helper function to assist in migrating from legacy Cryptography API (CAPI) to CNG.

8 Authentication

See Section 6.3 Operator Authentication.

9   Security Relevant Data Items

The Cryptographic Primitives Library crypto module uses the following security relevant data items:

Table 5

Security Relevant Data Item	Description
Symmetric encryption/decryption keys	Keys used for AES or Triple-DES encryption/decryption. Key sizes for AES are 128, 192, and 256 bits, and key sizes for Triple-DES are 192 and 128 bits.
HMAC keys	Keys used for HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512
Asymmetric DSA Public Keys	Keys used for the verification of DSA digital signatures. Key sizes are 2048 and 3072 bits.
Asymmetric DSA Private Keys	Keys used for the calculation of DSA digital signatures. Key sizes are 2048 and 3072 bits.
Asymmetric ECDSA Public Keys	Keys used for the verification of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
Asymmetric ECDSA Private Keys	Keys used for the calculation of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
Asymmetric RSA Public Keys	Keys used for the verification of RSA digital signatures. Key sizes are 2048 and 3072 bits. These keys can be produced using RSA Key Generation.
Asymmetric RSA Private Keys	Keys used for the calculation of RSA digital signatures. Key sizes are 2048 and 3072 bits. These keys can be produced using RSA Key Generation.
AES-CTR DRBG Seed	A secret value maintained internal to the module that provides the seed material for AES-CTR DRBG output
AES-CTR DRBG Entropy Input	A secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output
AES-CTR DRBG V	A secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output
AES-CTR DRBG key	A secret value maintained internal to the module that provides the entropy material for AES-CTR DRBG output
DH Private and Public values	Private and public values used for Diffie-Hellman key establishment. Key sizes are 2048 to 4096 bits.
ECDH Private and Public values	Private and public values used for EC Diffie-Hellman key establishment. Curve sizes are P-256, P-384, and P-521.
IKEv1 and IKEv2 DH shared secrets	Diffie-Hellman shared secret lengths are 2048 (SHA 256), 256 (SHA 256), and 384 (SHA 384).
TLS Derived Key	A key derived for the TLS protocol. Key size is 384 bits.

9.1   Access Control Policy

The Cryptographic Primitives Library crypto module allows controlled access to security relevant data items contained within it. The following table defines the access that a service has to each. The permissions are categorized as a set of four separate permissions: read (r), write (w), execute (x), delete

(d). If no permission is listed, the service has no access to the item. The User and Cryptographic Officer roles have the same access to keys so roles are not distinguished in the table.

Table 6

Cryptographic Primitives Library crypto module Service Access Policy	Symmetric encryption and decryption keys	HMAC keys	Asymmetric DSA Public Keys	Asymmetric DSA Private Keys	Asymmetric ECDSA Public keys	Asymmetric ECDSA Private keys	Asymmetric RSA Public Keys	Asymmetric RSA Private Keys	DH Public and Private values	ECDH Public and Private values	AES-CTR DRBG Seed, AES-CTR DRBG Entropy Input, AES-CTR DRBG V, & AES-CTR DRBG key	IKEv1 & IKEv2 DH Shared Secret	TLS Derived Key
Algorithm Providers and Properties													
Random Number Generation											x		
Key and Key-Pair Generation	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	x		
Key Entry and Output	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw			
Encryption and Decryption	x												
Hashing and Message Authentication		xw											
Signing and Verification			x	x	x	x	x	x			x		
Secret Agreement and Key Derivation									x	x	x	x	x
Show Status													
Self-Tests													
Zeroization	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd	wd

9.2   Key Material

Each time an application links with Cryptographic Primitives Library, the DLL is instantiated and no keys exist within it. The user application is responsible for importing keys into Cryptographic Primitives Library or using Cryptographic Primitives Library's functions to generate keys.

9.3   Key Generation

Cryptographic Primitives Library can create and use keys for the following algorithms: RSA, DSA, DH, ECDH, ECDSA, RC2, RC4, DES, Triple-DES, AES, and HMAC. However, RC2, RC4, and DES cannot be used in FIPS mode.

Random keys can be generated by calling the BCryptGenerateSymmetricKey() and BCryptGenerateKeyPair() functions. Random data generated by the BCryptGenRandom() function is provided to BCryptGenerateSymmetricKey() function to generate symmetric keys. DES, Triple-DES, AES,

RSA, ECDSA, DSA, DH, and ECDH keys and key-pairs are generated following the techniques given in SP 800-56Ar2 (Section 5.8.1).

Keys generated while not operating in the FIPS mode of operation (as described in section 2) cannot be used in FIPS mode, and vice versa.

9.4   Key Establishment

Cryptographic Primitives Library can use FIPS approved Diffie-Hellman key agreement (DH), Elliptic Curve Diffie-Hellman key agreement (ECDH), RSA key transport and manual methods to establish keys. Alternatively, the module can also use Approved KDFs to derive key material from a specified secret value or password.

Cryptographic Primitives Library can use the following FIPS approved key derivation functions (KDF) from the common secret that is established during the execution of DH and ECDH key agreement algorithms:

- BCRYPT_KDF_SP80056A_CONCAT. This KDF supports the Concatenation KDF as specified in SP 800-56A (Section 5.8.1).
- BCRYPT_KDF_HASH. This KDF supports FIPS approved SP 800-56A (Section 5.8), X9.63, and X9.42 key derivation.
- BCRYPT_KDF_HMAC. This KDF supports the IPsec IKEv1 key derivation that is non-Approved but is an allowed legacy implementation in FIPS mode when used to establish keys for IKEv1 as per scenario 4 of IG D.8.
- BCRYPT_KDF_TLS_PRF. This KDF supports the SSLv3.1 and TLSv1.0 key derivation that is non-Approved but is an allowed legacy implementation in FIPS mode when used to establish keys for SSLv3.1 or TLSv1.0 as specified in as per scenario 4 of IG D.8.

Cryptographic Primitives Library can use the following FIPS approved key derivation functions (KDF) from a key handle created from a specified secret or password:

- BCRYPT_SP800108_CTR_HMAC_ALGORITHM. This KDF supports the counter-mode variant of the KDF specified in SP 800-108 (Section 5.1) with HMAC as the underlying PRF.
- BCRYPT_SP80056A_CONCAT_ALGORITHM. This KDF supports the Concatenation KDF as specified in SP 800-56Ar2 (Section 5.8.1).
- BCRYPT_PBKDF2_ALGORITHM. This KDF supports the Password Based Key Derivation Function specified in SP 800-132 (Section 5.3).
- BCRYPT_CAPI_KDF_ALGORITHM. This KDF supports the proprietary KDF described at http://msdn.microsoft.com/library/windows/desktop/aa379916.aspx

9.4.1   NIST SP 800-132 Password Based Key Derivation Function (PBKDF)

There are two (2) options presented in NIST SP 800-132, pages 8 – 10, that are used to derive the Data Protection Key (DPK) from the Master Key. With the Cryptographic Primitives Library, it is up to the caller to select the option to generate/protect the DPK. For example, DPAPI uses option 2a. Cryptographic Primitives Library provides all the building blocks for the caller to select the desired option.

Cryptographic Primitives Library

The Cryptographic Primitives Library supports the following HMAC hash functions as parameters for PBKDF:

- SHA-1 HMAC
- SHA-256 HMAC
- SHA-384 HMAC
- SHA-512 HMAC

Keys derived from passwords, as shown in SP 800-132, may only be used in storage applications. In order to run in a FIPS Approved manner, strong passwords must be used and they may only be used for storage applications. The password/passphrase length is enforced by the caller of the PBKDF interfaces at the time the password/passphrase is created and not by this cryptographic module. (This module is not involved in the creation of any password/passphrase.)

For an example of usage, examine Boot Manager's support for BitLocker® encrypted volumes. In this case, for the password that is used in key derivation, 128 bits of entropy are generated from the system DRBG, then converted into 40 digits (3.2 bits of entropy per digit), which are then broken into groups of five digits that are each multiplied by 11 to create six digit groupings for parity/correctness checking on user entry. This password has 128-bit security. The upper bound for the probability of having this parameter guessed at random is 2^{-128}. This probability is not only based on the length of the password, but also the difficulty of guessing it. In accordance with SP 800-132 Section 5.1, the length of the salt, which is generated using a FIPS Approved Random Bit Generator, is 128 bits. The PBKDF iteration count (see SP 800-132 Section 5.2) is 2^{20}, which is 1,048,576.

9.4.2   NIST SP 800-38F AES Key Wrapping
As outlined in FIPS 140-2 IG, D.2 and D.9, AES key wrapping serves as a form of key transport, which in turn is a form of key establishment. This implementation of AES key wrapping is in accordance with NIST SP 800-38F Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping.

9.5   Key Entry and Output
Keys can be both exported and imported out of and into Cryptographic Primitives Library via BCryptExportKey(), BCryptImportKey(), and BCryptImportKeyPair() functions.

Symmetric key entry and output can also be done by exchanging keys using the recipient's asymmetric public key via BCryptSecretAgreement() and BCryptDeriveKey() functions.

Exporting the RSA private key by supplying a blob type of BCRYPT_PRIVATE_KEY_BLOB, BCRYPT_RSAFULLPRIVATE_BLOB, or BCRYPT_RSAPRIVATE_BLOB to BCryptExportKey() is not allowed in FIPS mode.

9.6   Key Storage
Cryptographic Primitives Library does not provide persistent storage of keys.

9.7   Key Archival

Cryptographic Primitives Library does not directly archive cryptographic keys. The Authenticated User may choose to export a cryptographic key (cf. "Key Entry and Output" above), but management of the secure archival of that key is the responsibility of the user.

9.8   Key Zeroization

All keys are destroyed and their memory location zeroized when the operator calls BCryptDestroyKey() or BCryptDestroySecret() on that key handle.

10 Self-Tests

10.1 Power-On Self-Tests

Cryptographic Primitives Library performs the following power-on (startup) self-tests when DllMain is called by the operating system.

- HMAC (SHA-1, SHA-256, and SHA-512) Known Answer Tests
- Triple-DES encrypt/decrypt ECB Known Answer Tests
- AES-128 encrypt/decrypt ECB Known Answer Tests
- AES-128 encrypt/decrypt CCM Known Answer Tests
- AES-128 encrypt/decrypt CBC Known Answer Tests
- AES-128 CMAC Known Answer Test
- AES-128 encrypt/decrypt GCM Known Answer Tests
- XTS-AES encrypt/decrypt Known Answer Tests
- RSA sign/verify Known Answer Tests using RSA_SHA256_PKCS1 signature generation and verification
- DSA sign/verify tests with 2048-bit key
- ECDSA sign/verify Known Answer Tests on P256 curve
- DH secret agreement Known Answer Test with 2048-bit key
- ECDH secret agreement Known Answer Test on P256 curve
- SP 800-56A concatenation KDF Known Answer Tests (same as Diffie-Hellman KAT)
- SP 800-90A AES-256 based counter mode random generator Known Answer Tests (instantiate, generate and reseed)
- SP 800-108 KDF Known Answer Test
- SP 800-132 PBKDF Known Answer Test

In all cases for any failure of a power-on (startup) self-test, Cryptographic Primitives Library DllMain fails to return the STATUS_SUCCESS status to the operating system. The only way to recover from the failure of a power-on (startup) self-test is to attempt to reload the Cryptographic Primitives Library, which will rerun the self-tests, and will only succeed if the self-tests pass.

Cryptographic Primitives Library

10.2 Conditional Self-Tests

Cryptographic Primitives Library performs pair-wise consistency checks upon each invocation of RSA, ECDH, DSA, and ECDSA key-pair generation and import as defined in FIPS 140-2. SP 800-56A conditional self-tests are also performed. A Continuous Random Number Generator Test (CRNGT) is used for the random number generators and the Deterministic Random Bit Generator (DRBG) of this cryptographic module, which includes the SP 800-90A AES-256 CTR DRBG. There is also a CRNGT for the entropy source of the DRBGs. All approved and non-approved DRBGs have a CRNGT. If the conditional self-test fails, the module will not load and status will be returned. If the status is not STATUS_SUCCESS, then that is the indicator a conditional self-test failed.

- CRNGT for SP 800-90A AES-CTR DRBG
- Pairwise consistency tests for DSA, ECDSA, and RSA key generations
- Pairwise consistency tests for Diffie-Hellman and EC Diffie-Hellman prime value generation
- Assurances for SP 800-56A (According to sections 5.5.2, 5.6.2, and 5.6.3 of the standard)
- DRBG health test for SP 800-90A AES-CTR

11 Design Assurance

The secure installation, generation, and startup procedures of this cryptographic module are part of the overall operating system secure installation, configuration, and startup procedures for the Windows 10 OEs. The various methods of delivery and installation for each product are listed in the following table.

Table 7

Product	Delivery and Installation Method
Windows 10, Windows 10 Pro, Windows 10 Enterprise, Windows Enterprise LTSB, Windows Server 2016 Standard, Windows Server 2016 Datacenter	• Pre-installed on the computer by OEM • Download that updates to Windows 10 • Enterprise IT deployment
Surface Book, Surface Pro 4, Surface Pro 3, Surface 3, Lumia 950, Windows Storage Server 2016	• Pre-installed by the OEM (Microsoft)

After the operating system has been installed, it must be configured by enabling the "System cryptography: Use FIPS compliant algorithms for encryption, hashing, and signing" policy setting followed by restarting the system. This procedure is all the crypto officer and user behavior necessary for the secure operation of this cryptographic module.

An inspection of authenticity of the physical medium can be made by following the guidance at this Microsoft web site: https://www.microsoft.com/en-us/howtotell/default.aspx

The installed version of Windows 10 OEs must be verified to match the version that was validated. See Appendix A for details on how to do this.

For Windows Updates, the client only accepts binaries signed by Microsoft certificates. The Windows Update client only accepts content whose SHA-2 hash matches the SHA-2 hash specified in the metadata. All metadata communication is done over a Secure Sockets Layer (SSL) port. Using SSL ensures that the client is communicating with the real server and so prevents a spoof server from sending the client harmful requests. The version and digital signature of new cryptographic module releases must be verified to match the version that was validated. See Appendix A for details on how to do this.

12 Mitigation of Other Attacks

The following table lists the mitigations of other attacks for this cryptographic module:

Table 8

Algorithm	Protected Against	Mitigation	Comments
SHA1	Timing Analysis Attack	Constant Time Implementation	
	Cache Attack	Memory Access pattern is independent of any confidential data	
SHA2	Timing Analysis Attack	Constant Time Implementation	
	Cache Attack	Memory Access pattern is independent of any confidential data	
Triple-DES	Timing Analysis Attack	Constant Time Implementation	
AES	Timing Analysis Attack	Constant Time Implementation	
	Cache Attack	Memory Access pattern is independent of any confidential data	Protected Against Cache attacks only when used with AES NI

13 Security Levels

The security level for each FIPS 140-2 security requirement is given in the following table.

Table 9

Security Requirement	Security Level
Cryptographic Module Specification	1
Cryptographic Module Ports and Interfaces	1
Roles, Services, and Authentication	1
Finite State Model	1
Physical Security	NA
Operational Environment	1
Cryptographic Key Management	1
EMI/EMC	1
Self-Tests	1
Design Assurance	2
Mitigation of Other Attacks	1

14 Additional Details

For the latest information on Microsoft Windows, check out the Microsoft web site at:

https://www.microsoft.com/en-us/windows

For more information about FIPS 140 validations of Microsoft products, please see:

https://technet.microsoft.com/en-us/library/cc750357.aspx

15 Appendix A – How to Verify Windows Versions and Digital Signatures

15.1 How to Verify Windows Versions

The installed version of Windows 10 OEs must be verified to match the version that was validated using the following method:

1.   In the Search box type "cmd" and open the Command Prompt desktop app.
2.   The command window will open.
3.   At the prompt, enter "ver".
4.   The version information will be displayed in a format like this:
           `Microsoft Windows [Version 10.0.xxxxx]`

If the version number reported by the utility matches the expected output, then the installed version has been validated to be correct.

15.2 How to Verify Windows Digital Signatures

After performing a Windows Update that includes changes to a cryptographic module, the digital signature and file version of the binary executable file must be verified. This is done like so:

1. Open a new window in Windows Explorer.
2. Type "C:\Windows\" in the file path field at the top of the window.
3. Type the cryptographic module binary executable file name (for example, "CNG.SYS") in the search field at the top right of the window, then press the Enter key.
4. The file will appear in the window.
5. Right click on the file's icon.
6. Select Properties from the menu and the Properties window opens.
7. Select the Details tab.
8. Note the File version Property and its value, which has a number in this format: xx.x.xxxxx.xxxx.
9. If the file version number matches one of the version numbers that appear at the start of this security policy document, then the version number has been verified.
10. Select the Digital Signatures tab.
11. In the Signature list, select the Microsoft Windows signer.
12. Click the Details button.
13. Under the Digital Signature Information, you should see: "This digital signature is OK." If that condition is true, then the digital signature has been verified.