

Security Policy for FIPS 140-2 Validation

Cryptographic Primitives Library
(bcryptprimitives.dll and ncryptsslp.dll) in
Microsoft Windows 8.1 Enterprise
Windows Server 2012 R2
Windows Storage Server 2012 R2
Surface Pro 2
Surface Pro
Surface 2
Surface
Windows RT 8.1
Windows Phone 8.1
Windows Embedded 8.1 Industry Enterprise
StorSimple 8000 Series

DOCUMENT INFORMATION

Version Number Updated On	1.11
	April 24, 2015

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. This work is licensed under the Creative Commons Attribution-NoDerivs-NonCommercial License (which allows redistribution of the work). To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd-nc/1.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2015 Microsoft Corporation. All rights reserved.

Microsoft, Windows, the Windows logo, Windows Server, and BitLocker are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

CHANGE HISTORY

Date	Version	Updated By	Change
31 OCT 2013	0.1	Tim Myers	First Draft
29 APR 2014		Tim Myers	Minor change: updated DSS from FIPS 186-3 to 186-4
7 MAY 2014	0.2	Tim Myers	Second Draft; FIPS 186-4 and FIPS 180-4 updates; processor name updates
1 JUL 2014	0.3	Tim Myers	Third Draft; added Security Levels Table and cryptographic algorithm certificate numbers
10 JUL 2014	1.0	Tim Myers	First Release to Validators; includes all algorithm and module certificate numbers
18 JUL 2014	1.1	Tim Myers	Update platforms; consolidate services
12 DEC 2014	1.2	Tim Myers	Address CMVP comments; update platform names
17 DEC 2014	1.3	Tim Myers	Update IG G.5 platforms; update binary versions; add StorSimple
22 JAN 2015	1.4	Tim Myers	Address CMVP comments
29 JAN 2015	1.5	Tim Myers	Update Design Assurance
20 FEB 2015	1.6	Tim Myers	Add StorSimple to Validated Platforms
26 FEB 2015	1.7	Tim Myers	Clarify DH sizes; add KDF testing statement; add GCM note
9 MAR 2015	1.8	Tim Myers	Prepare for publication
17 MAR 2015	1.9	Tim Myers	Reorder StorSimple 8100 OE description; fix typos in Section 10
3 APR 2015	1.10	Tim Myers	Modify AES-GCM encryption
24 APR 2015	1.11	Tim Myers	Correct section 2.2 to agree with section 2.1

TABLE OF CONTENTS

<u>1</u>	<u>INTRODUCTION</u>	<u>7</u>
1.1	LIST OF CRYPTOGRAPHIC MODULE BINARY EXECUTABLES.....	7
1.2	BRIEF MODULE DESCRIPTION.....	7
1.3	VALIDATED PLATFORMS	7
1.4	CRYPTOGRAPHIC BOUNDARY	9
<u>2</u>	<u>SECURITY POLICY</u>	<u>10</u>
2.1	FIPS 140-2 APPROVED ALGORITHMS.....	11
2.2	NON-APPROVED ALGORITHMS	12
2.3	CRYPTOGRAPHIC BYPASS	12
2.4	MACHINE CONFIGURATIONS.....	13
<u>3</u>	<u>OPERATIONAL ENVIRONMENT.....</u>	<u>13</u>
<u>4</u>	<u>INTEGRITY CHAIN OF TRUST.....</u>	<u>13</u>
<u>5</u>	<u>PORTS AND INTERFACES</u>	<u>13</u>
5.1	EXPORT FUNCTIONS.....	13
5.2	CNG PRIMITIVE FUNCTIONS	14
5.3	CONTROL INPUT INTERFACE.....	15
5.4	STATUS OUTPUT INTERFACE	15
5.5	DATA OUTPUT INTERFACE.....	15
5.6	DATA INPUT INTERFACE	15
<u>6</u>	<u>SPECIFICATION OF ROLES</u>	<u>16</u>
6.1	MAINTENANCE ROLES	16
6.2	MULTIPLE CONCURRENT INTERACTIVE OPERATORS.....	16
6.3	OPERATOR AUTHENTICATION	16
<u>7</u>	<u>SERVICES.....</u>	<u>16</u>
7.1	ALGORITHM PROVIDERS AND PROPERTIES	16

7.1.1	BCRYPTOPENALGORITHMPROVIDER.....	16
7.1.2	BCRYPTCLOSEALGORITHMPROVIDER.....	16
7.1.3	BCRYPTSETPROPERTY	17
7.1.4	BCRYPTGETPROPERTY.....	17
7.1.5	BCRYPTFREEBUFFER	17
7.2	RANDOM NUMBER GENERATION.....	17
7.2.1	BCRYPTGENRANDOM	17
7.3	KEY AND KEY-PAIR GENERATION	18
7.3.1	BCRYPTGENERATESYMMETRICKEY	18
7.3.2	BCRYPTGENERATEKEYPAIR	18
7.3.3	BCRYPTFINALIZEKEYPAIR.....	19
7.3.4	BCRYPTDUPLICATEKEY	19
7.3.5	BCRYPTDESTROYKEY.....	19
7.4	KEY ENTRY AND OUTPUT	19
7.4.1	BCRYPTIMPORTKEY	19
7.4.2	BCRYPTIMPORTKEYPAIR.....	20
7.4.3	BCRYPTEXPORTKEY	21
7.5	ENCRYPTION AND DECRYPTION	23
7.5.1	BCRYPTENCRYPT	23
7.5.2	BCRYPTDECRYPT	24
7.6	HASHING AND MESSAGE AUTHENTICATION	25
7.6.1	BCRYPTCREATEHASH	25
7.6.2	BCRYPTHASHDATA.....	26
7.6.3	BCRYPTDUPLICATEHASH	26
7.6.4	BCRYPTFINISHHASH.....	26
7.6.5	BCRYPTDESTROYHASH	27
7.7	SIGNING AND VERIFICATION	27
7.7.1	BCRYPTSIGNHASH.....	27
7.7.2	BCRYPTVERIFYSIGNATURE	28
7.8	SECRET AGREEMENT AND KEY DERIVATION.....	29
7.8.1	BCRYPTSECRETAGREEMENT	29
7.8.2	BCRYPTDERIVEKEY	29
7.8.3	BCRYPTDESTROYSECRET	30
7.8.4	BCRYPTKEYDERIVATION	30
7.9	DEPRECATION	31
7.9.1	BIT STRENGTHS OF DH AND ECDH.....	31
7.9.2	SHA-1.....	31
7.10	SHOW STATUS SERVICES.....	31
7.11	SELF-TEST SERVICES.....	32
7.12	SERVICE INPUTS / OUTPUTS	32

<u>8</u>	<u>AUTHENTICATION</u>	<u>32</u>
<u>9</u>	<u>CRYPTOGRAPHIC KEY MANAGEMENT</u>	<u>32</u>
9.1	ACCESS CONTROL POLICY	33
9.2	KEY MATERIAL	34
9.3	KEY GENERATION	34
9.4	KEY ESTABLISHMENT	34
9.4.1	NIST SP 800-132 PASSWORD BASED KEY DERIVATION FUNCTION (PBKDF)	35
9.5	KEY ENTRY AND OUTPUT	35
9.6	KEY STORAGE	36
9.7	KEY ARCHIVAL	36
9.8	KEY ZEROIZATION	36
<u>10</u>	<u>SELF-TESTS</u>	<u>37</u>
10.1	POWER-ON SELF-TESTS	37
10.2	CONDITIONAL SELF-TESTS	37
<u>11</u>	<u>DESIGN ASSURANCE</u>	<u>38</u>
<u>12</u>	<u>MITIGATION OF OTHER ATTACKS</u>	<u>39</u>
<u>13</u>	<u>SECURITY LEVELS</u>	<u>40</u>
<u>14</u>	<u>ADDITIONAL DETAILS</u>	<u>40</u>
<u>15</u>	<u>APPENDIX A – HOW TO VERIFY WINDOWS VERSIONS AND DIGITAL SIGNATURES</u>	<u>41</u>
15.1	HOW TO VERIFY WINDOWS VERSIONS	41
15.2	HOW TO VERIFY WINDOWS DIGITAL SIGNATURES	41

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 Microsoft Windows 8.1 Enterprise, Windows Server 2012 R2, Windows Storage Server 2012 R2, Surface Pro 2, Surface Pro, Surface 2, Surface, Windows RT 8.1, Windows Phone 8.1, Windows Embedded 8.1 Industry Enterprise, and StorSimple 8000 Series (herein referred to as Windows 8.1 OEs).

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.

1.1 List of Cryptographic Module Binary Executables

BCRYPTPRIMITIVES.DLL – Versions 6.3.9600 and 6.3.9600.17031 for Windows 8.1 OEs

NCryptSSLP.DLL – Versions 6.3.9600 and 6.3.9600.17031 for Windows 8.1 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 component listed in Section 1.1 was validated using the following machine configurations:

1. Microsoft Windows 8.1 Enterprise (x86) running on a Dell PowerEdge SC440 without AES-NI;
2. Microsoft Windows Embedded 8.1 Industry Enterprise (x86) running on a Dell PowerEdge SC440 without AES-NI;
3. Microsoft Windows 8.1 Enterprise (x86) running on a Dell Dimension E521 without AES-NI;
4. Microsoft Windows Embedded 8.1 Industry Enterprise (x86) running on a Dell Dimension E521 without AES-NI;
5. Microsoft Windows 8.1 Enterprise (x86) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
6. Microsoft Windows Embedded 8.1 Industry Enterprise (x86) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
7. Microsoft Windows 8.1 Enterprise (x86) running on an HP Compaq Pro 6305 with AES-NI;
8. Microsoft Windows Embedded 8.1 Industry Enterprise (x86) running on an HP Compaq Pro 6305 with AES-NI;
9. Microsoft Windows 8.1 Enterprise (x64) running on a Dell PowerEdge SC440 without AES-NI;

10. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on a Dell PowerEdge SC440 without AES-NI;
11. Microsoft Windows Server 2012 R2 (x64) running on a Dell PowerEdge SC440 without AES-NI;
12. Microsoft Windows Storage Server 2012 R2 (x64) running on a Dell PowerEdge SC440 without AES-NI;
13. Microsoft Windows 8.1 Enterprise (x64) running on a Dell Dimension E521 without AES-NI;
14. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on a Dell Dimension E521 without AES-NI;
15. Microsoft Windows Server 2012 R2 (x64) running on a Dell Dimension E521 without AES-NI;
16. Microsoft Windows Storage Server 2012 R2 (x64) running on a Dell Dimension E521 without AES-NI;
17. Microsoft Windows 8.1 Enterprise (x64) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
18. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
19. Microsoft Windows Server 2012 R2 (x64) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
20. Microsoft Windows Storage Server 2012 R2 (x64) running on an Intel Core i7 with AES-NI running on an Intel Maho Bay;
21. Microsoft Windows 8.1 Pro (x64) running on an Intel x64 Processor with AES-NI running on a Microsoft Surface Pro;
22. Microsoft Windows 8.1 Pro (x64) running on an Intel i5 with AES-NI running on a Microsoft Surface Pro 2;
23. Microsoft Windows 8.1 Enterprise (x64) running on an HP Compaq Pro 6305 with AES-NI;
24. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on an HP Compaq Pro 6305 with AES-NI;
25. Microsoft Windows Server 2012 R2 (x64) running on an HP Compaq Pro 6305 with AES-NI;
26. Microsoft Windows Storage Server 2012 R2 (x64) running on an HP Compaq Pro 6305 with AES-NI;
27. Microsoft Windows 8.1 RT (ARMv7 Thumb-2) running on an NVIDIA Tegra 3 Tablet;
28. Microsoft Windows 8.1 RT (ARMv7 Thumb-2) running on a Microsoft Surface RT;
29. Microsoft Windows 8.1 RT (ARMv7 Thumb-2) running on a Microsoft Surface 2;
30. Microsoft Windows 8.1 RT (ARMv7 Thumb-2) running on a Qualcomm Tablet;
31. Microsoft Windows Phone 8.1 (ARMv7 Thumb-2) running on a Qualcomm Snapdragon S4 running on a Windows Phone 8.1;
32. Microsoft Windows Phone 8.1 (ARMv7 Thumb-2) running on a Qualcomm Snapdragon 400 running on a Windows Phone 8.1;
33. Microsoft Windows Phone 8.1 (ARMv7 Thumb-2) running on a Qualcomm Snapdragon 800 running on a Windows Phone 8.1;
34. Microsoft Windows 8.1 Enterprise (x64) running on a Dell Inspiron 660s without AES-NI and with PCLMULQDQ and SSSE 3;
35. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on a Dell Inspiron 660s without AES-NI and with PCLMULQDQ and SSSE 3;
36. Microsoft Windows Server 2012 R2 (x64) running on a Dell Inspiron 660s without AES-NI and with PCLMULQDQ and SSSE 3;

37. Microsoft Windows Storage Server 2012 R2 (x64) running on a Dell Inspiron 660s without AES-NI and with PCLMULQDQ and SSSE 3;
38. Microsoft Windows 8.1 Enterprise (x64) running on an Intel Core i5 with AES-NI and with PCLMULQDQ and SSSE 3 running on a Microsoft Surface Pro 2;
39. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on an Intel Core i7 with AES-NI and with PCLMULQDQ and SSSE 3 running on an Intel Maho Bay;
40. Microsoft Windows Server 2012 R2 (x64) running on an Intel Core i7 with AES-NI and with PCLMULQDQ and SSSE 3 running on an Intel Maho Bay;
41. Microsoft Windows Storage Server 2012 R2 (x64) running on an Intel Core i7 with AES-NI and with PCLMULQDQ and SSSE 3 running on an Intel Maho Bay;
42. Microsoft Windows 8.1 Enterprise (x64) running on an HP Compaq Pro 6305 with AES-NI and with PCLMULQDQ and SSSE 3;
43. Microsoft Windows Embedded 8.1 Industry Enterprise (x64) running on an HP Compaq Pro 6305 with AES-NI and with PCLMULQDQ and SSSE 3;
44. Microsoft Windows Server 2012 R2 (x64) running on an HP Compaq Pro 6305 with AES-NI and with PCLMULQDQ and SSSE 3;
45. Microsoft Windows Storage Server 2012 R2 (x64) running on an HP Compaq Pro 6305 with AES-NI and with PCLMULQDQ and SSSE 3;
46. Windows Server 2012 R2 (x64) running on a Microsoft StorSimple 8100 with an Intel Xeon E5-2648L without AES-NI;
47. Windows Server 2012 R2 (x64) running on a Microsoft StorSimple 8100 with an Intel Xeon E5-2648L with AES-NI

The Cryptographic Primitives Library component maintains FIPS 140-2 validation compliance (according to FIPS 140-2 PUB Implementation Guidance G.5) on the following platforms:

x86 Microsoft Windows 8.1
x86 Microsoft Windows 8.1 Pro

x64 Microsoft Windows 8.1
x64 Microsoft Windows Server 2012 R2 Datacenter

x64-AES-NI Microsoft Windows 8.1
x64-AES-NI Microsoft Windows Server 2012 R2 Datacenter

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 8.1 OEs.
- Cryptographic Primitives Library operates in FIPS mode of operation only when used with the FIPS approved version of Windows 8.1 OEs Code Integrity (ci.dll) validated to FIPS 140-2 under Cert. #2355, 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. #2356) operating in FIPS mode, required for entropy input (see section on BCryptGenRandom).
- Windows 8.1 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 only when one of the following DWORD registry values is set to 1:
 - HKLM\SYSTEM\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy\Enabled
 - HKLM\SYSTEM\CurrentControlSet\Policies\Microsoft\Cryptography\Configuration\SelfTestAlgorithms
- The registry security policy settings can be observed with the regedit tool to determine whether the module is in FIPS mode.
- 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 8.1 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 illustrates the master components of the Cryptographic Primitives Library module:

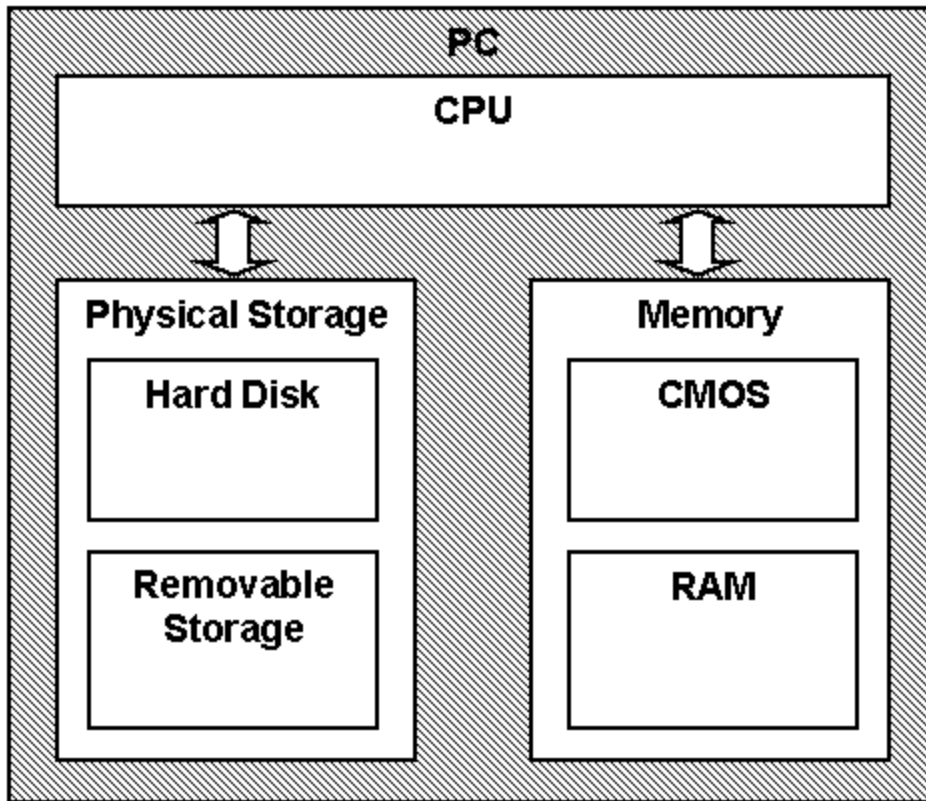


Figure 1 Master components of Cryptographic Primitives Library module

2.1 FIPS 140-2 Approved Algorithms

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

- FIPS 180-4 SHS (SHA-256, SHA-384, and SHA-512 hash) (Cert. #2373)
- FIPS 180-4 SHA-1, SHA-256, SHA-384, SHA-512 HMAC (Cert. #1773)
- Triple-DES (2 key¹ and 3 key) in ECB, CBC, CFB8 and CFB64 modes (Cert. #1692)
- AES-128, AES-192, and AES-256 in ECB, CBC, CFB8, CFB128, and CTR modes (Cert. #2832)
- AES-128, AES-192, and AES-256 in CCM, GCM, GMAC, and CMAC modes (Cert#2832)
- AES-128, AES-192, and AES-256 GCM decryption (Cert#2832)
- FIPS 186-4 RSA (RSASSA-PKCS1-v1_5 and RSASSA-PSS) digital signatures (Certs. #1493 and #1519) and FIPS 186-4 RSA key-pair generation (Cert. #1487)
- FIPS 186-4 DSA (Cert. #855)
- FIPS 186-4 ECDSA with the following NIST curves: P-256, P-384, P-521 (Cert. #505)
- KAS – SP 800-56A (Cert. #47) Diffie-Hellman Key Agreement; key establishment methodology provides 80 bits² and 112 bits of encryption strength

¹ Two-key Triple-DES is *restricted* and *legacy-use* according to NIST SP 800-131A. Users should start transitioning away from this algorithm to better, stronger choices.

² DH with 80 bits of encryption strength is no longer allowed in FIPS mode. The minimum requirement has increased to 112 bits effective January 1, 2014.

- KAS – SP 800-56A (Cert. #47) EC Diffie-Hellman Key Agreement; key establishment methodology provides between 128 and 256-bits of encryption strength
- SP 800-90 AES-256 counter mode DRBG (Cert. #489)
- SP 800-90 Dual EC DRBG (Cert. #523)
- SP 800-108 Key Derivation Function (KDF) (Cert. #30)
- SP 800-132 KDF (also known as PBKDF) (vendor-affirmed)
- SP 800-135 IKEv1, IKEv2, and TLS KDF primitives (Cert. #323)³

2.2 Non-Approved Algorithms

- Cryptographic Primitives Library implements the SHA-1 hash, which is disallowed for use in digital signature generation after December 31, 2013. It can be used for digital signature verification legacy-use. Its use is Acceptable for non-digital signature generation applications.
- Cryptographic Primitives Library implements RSA 1024-bits for digital signature generation, which is disallowed after December 31, 2013. 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 after December 31, 2013. FB (p=2048, q=224) and FC (p=2048, q=256) are Acceptable.
- Cryptographic Primitives Library has a non-approved algorithm implementation of AES GCM encryption.
- Cryptographic Primitives Library supports the following non-Approved algorithms allowed for use in FIPS mode:
 - AES Key Wrap (AES Cert. #2832, key wrapping; key establishment methodology provides between 128 and 256 bits of encryption strength)
 - MD5 and HMAC MD5 (allowed in TLS and EAP-TLS)
- Cryptographic Primitives Library also supports the following non FIPS 140-2 approved algorithms, though these algorithms may not be used when operating the module in a FIPS compliant manner.
 - RSA encrypt/decrypt
 - RC2, RC4, MD2, MD4⁴
 - DES in ECB, CBC, CFB8 and CFB64 modes
 - Legacy CAPI KDF (proprietary)
 - NDRNG

2.3 Cryptographic Bypass

Cryptographic bypass is not supported by Cryptographic Primitives Library.

³ This cryptographic module supports the TLS, IKEv1, and IKEv2 protocols with SP 800-135 rev 1 KDF primitives, however, the protocols proper have not been reviewed or tested by the NIST CAVP and CMVP.

⁴ Applications may not use any of these non-FIPS algorithms if they need to be FIPS compliant. To operate the module in a FIPS compliant manner, applications must only use FIPS-approved algorithms.

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 8.1 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, based on verification of SHA-256 page hashes and an RSA signature with a 2048-bit key using SHA-256 as the underlying hash (Cert. #2373 for SHA-256 and Cert. #1494 for RSA signature).

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:

<http://msdn.microsoft.com/en-us/library/aa375534.aspx>.

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 8.1 OEs, available at:

[http://msdn.microsoft.com/en-us/library/windows/desktop/aa376210\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa376210(v=vs.85).aspx)

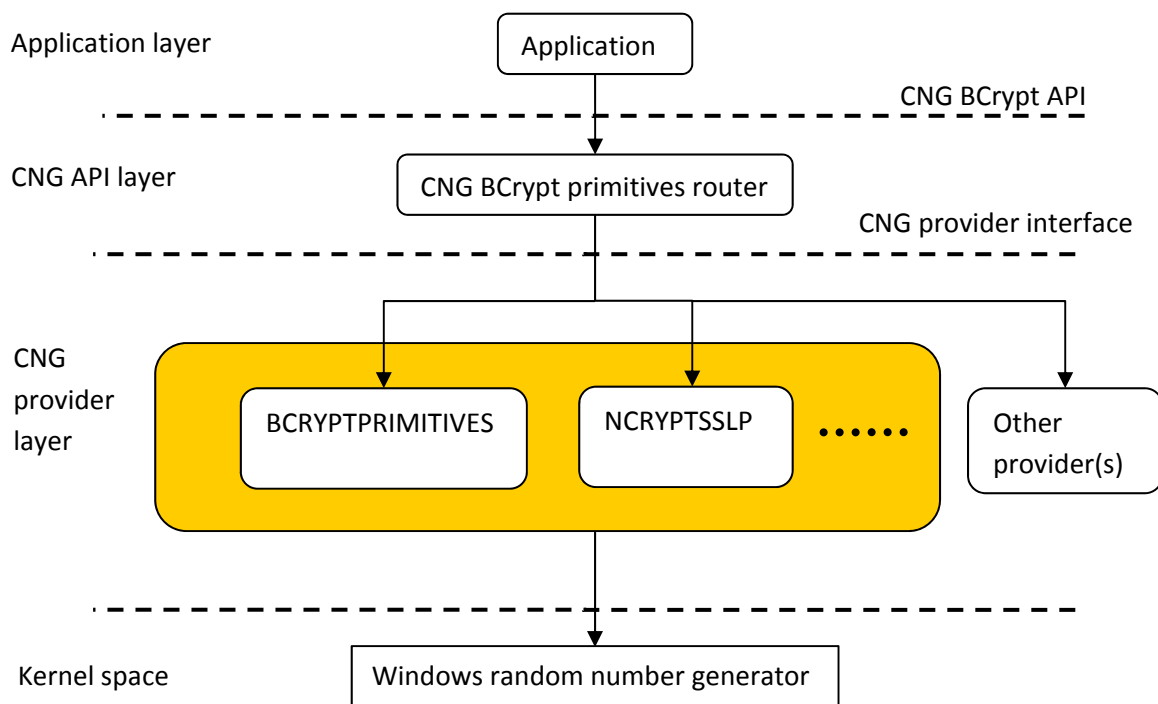


Figure 2 Relationships between Cryptographic Primitives Library and other components – cryptographic boundary highlighted in gold.

The following functions are exported by Cryptographic Primitives Library:

- GetAsymmetricEncryptionInterface
- GetCipherInterface
- GetHashInterface
- GetRngInterface
- GetSecretAgreementInterface
- GetSignatureInterface

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
- BCryptDecrypt
- BCryptDeriveKey
- BCryptDestroyHash

- BCryptDestroyKey
- BCryptDestroySecret
- BCryptDuplicateHash
- BCryptDuplicateKey
- BCryptEncrypt
- BCryptExportKey
- BCryptFinalizeKeyPair
- BCryptFinishHash
- BCryptFreeBuffer
- BCryptGenerateKeyPair
- BCryptGenerateSymmetricKey
- BCryptGenRandom
- BCryptGetProperty
- BCryptHashData
- BCryptImportKey
- BCryptImportKeyPair
- BCryptKeyDerivation
- BCryptOpenAlgorithmProvider
- BCryptSecretAgreement
- BCryptSetProperty
- BCryptSignHash
- BCryptVerifySignature

5.3 Control Input Interface

The Control Input Interface for Cryptographic Primitives Library consists of the CNG primitive functions listed in Section 5.2. 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 5.2. 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 5.2.

5.6 Data Input Interface

The Data Input Interface for Cryptographic Primitives Library also consists of the CNG primitive functions listed in Section 5.2. 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.

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.

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.

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 <http://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,  
    PCHAR 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,  
    PCHAR 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,  
    PCHAR pbBuffer,  
    ULONG cbBuffer,  
    ULONG dwFlags);
```

The `BCryptGenRandom()` function fills a buffer with random bytes. `BCRYPTPRIMITVES.DLL` implements two random number generation algorithms:

- **BCRYPT_RNG_ALGORITHM.** This is the AES-256 counter mode based random generator as defined in SP 800-90.
- **BCRYPT_RNG_DUAL_EC_ALGORITHM.** This is the dual elliptic curve based random generator as defined in SP 800-90.

During the function initialization, a seed is obtained from the output of an in-kernel random number generator. This RNG, which exists beyond the cryptographic boundary, provides the necessary entropy for the user-level RNGs available through this function. A description of the entropy collection is documented with the `SystemPng` 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,
    PCHAR pbKeyObject,
    ULONG cbKeyObject,
    PCHAR 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,
    PCHAR 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,
    PCHAR pbKeyObject,
    ULONG cbKeyObject,
    PCHAR 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,  
    PCHAR 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 *pblInput* 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.

pblInput [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 *pblInput* 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,  
    PCHAR 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,
    PCHAR pbInput,
    ULONG cbInput,
    VOID *pPaddingInfo,
    PCHAR pbIV,
    ULONG cbIV,
    PCHAR 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)⁵ 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 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.

⁵ The IV being passed into the BCryptEncrypt() API needs to be generated by a validated cryptographic module or an approved DRBG.

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,
    PCHAR pbInput,
    ULONG cbInput,
    VOID *pPaddingInfo,
    PCHAR pbIV,
    ULONG cbIV,
    PCHAR 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.

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,
    PCHAR pbHashObject,
    ULONG cbHashObject,
    PCHAR 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,
    PCHAR 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,
    PCHAR 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.

7.7 Signing and Verification

7.7.1 BCryptSignHash

```
NTSTATUS WINAPI BCryptSignHash(  
    BCRYPT_KEY_HANDLE hKey,  
    VOID *pPaddingInfo,  
    PCHAR pbInput,  
    ULONG cbInput,  
    PCHAR 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 currently *deprecated* for digital signature generation and will be *disallowed* after the end of 2013. SHA-1 is currently *legacy-use* for digital signature verification.

7.7.2 BCRYPTVerifySignature

```
NTSTATUS WINAPI BCRYPTVerifySignature(
    BCRYPT_KEY_HANDLE hKey,
    VOID *pPaddingInfo,
    PCHAR pbHash,
    ULONG cbHash,
    PCHAR 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 currently *deprecated* for digital signature generation and will be *disallowed* after the end of 2013. 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,
    PCHAR               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_PR_F (parameters in pParameterList: KDF_TLS_PR_F_LABEL, KDF_TLS_PR_F_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://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.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 80 bits for FA, 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) PCHAR 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.

7.9 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 is considered deprecated, and will be disallowed starting January 1, 2014. On that date, only security strength of at least 112 bits will be 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 will change to 2048 to 4096 after 2013.

7.9.2 SHA-1

From 2011 through 2013, SHA-1 can be used in a deprecated mode for use in digital signature generation. On Jan. 1, 2014, SHA-1 will no longer be allowed for digital signature generation, and it will be 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.

8 Authentication

See Section 6.3 Operator Authentication.

9 Cryptographic Key Management

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

Security Relevant Data Item	Description
Symmetric encryption/decryption keys	Keys used for AES or TDEA encryption/decryption. Key sizes for AES are 128, 192, and 256 bits, and key sizes for TDES are 168 and 112 bits.
HMAC keys	Keys used for HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512
DSA Public Keys	Keys used for the verification of DSA digital signatures. Key sizes are 2048 and 3072 bits.
DSA Private Keys	Keys used for the calculation of DSA digital signatures. Key sizes are 2048 and 3072 bits.
ECDSA Public Keys	Keys used for the verification of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
ECDSA Private Keys	Keys used for the calculation of ECDSA digital signatures. Curve sizes are P-256, P-384, and P-521.
RSA Public Keys	Keys used for the verification of RSA digital signatures. Key sizes are 2048 and 3072 bits.
RSA Private Keys	Keys used for the calculation of RSA digital signatures. Key sizes are 2048 and 3072 bits.
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.
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
DUAL EC DRBG Seed	A secret value maintained internal to the module that provides the seed material for DUAL EC DRBG output
DUAL EC DRBG Entropy Input	A secret value maintained internal to the module that provides the entropy material for DUAL EC DRBG output
DUAL EC DRBG V	A secret value maintained internal to the module that provides the entropy material for DUAL EC DRBG output
DUAL EC DRBG key	A secret value maintained internal to the module that provides the entropy material for DUAL EC DRBG output

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.

Cryptographic Primitives Library crypto module Service Access Policy	Symmetric encryption and decryption keys	HMAC keys	DSA Public Keys	DSA Private Keys	ECDSA public keys	ECDSA Private keys	RSA Public Keys	RSA Private Keys	DH Public and Private values	ECDH Public and Private values	DRBG Seeds and CSPs
Cryptographic Module Power Up and Power Down											
Key Formatting	w										
Random Number Generation											x
Data Encryption and Decryption	x										
Hashing		xw									
Acquiring a Table of Pointers to BCryptXXX Functions											
Algorithm Providers and Properties											
Key and Key-Pair Generation	wd	wd	wd	w d	w d	w d	w d	w d	wd	wd	x
Key Entry and Output	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
Signing and Verification			x	x	x	x	x	x			x
Secret Agreement and Key Derivation									x	x	x

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. All algorithms can be used in FIPS mode, except: DES, RC2, and RC4.

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 section 7.2.

The module generates cryptographic keys whose strengths are modified by available entropy.

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_PR`F. 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 800-56A (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 Kernel Mode Cryptographic Primitives Library, it is up to the caller to select the option to generate/protect the DPK. For example, DPAPI uses option 2a. Kernel Mode Cryptographic Primitives Library provides all the building blocks for the caller to select the desired option.

The Kernel Mode 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, it is up to the user and application to pick strong passwords and use them only for storage applications. The password/passphrase length is enforced by the caller of the PBKDF interfaces and not the cryptographic module. In order to run in a FIPS approved manner, the password must be chosen in accordance with the guidelines in NIST SP 800-63 Electronic Authentication Guideline and SP 800-118 DRAFT Guide to Enterprise Password Management. The upper bound for the probability of having the password guessed at random is to be computed following the SP 800-63 and SP 800-118 guidelines. The decision for the minimum length of a password used for key derivation is to be based on the SP 800-63 and SP 800-118 guidelines.

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

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.

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 RNG test (CRNGT) is used for the random number generators and the Deterministic Random Bit Generator (DRBG) of this cryptographic module, which includes the SP 800-90 AES-256 CTR DRBG and the Dual Elliptic Curve DRBG. There is also a CRNGT for the entropy source of the DRBGs.

- SP800-90 AES-CTR DRBG
- SP800-90 Dual-EC 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 SP800-56A (According to sections 5.5.2, 5.6.2, and 5.6.3 of the standard)
- DRBG health test for SP800-90 AES-CTR
- DRBG health test for Dual-EC DRBGs

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 8.1 OEs. The various methods of delivery and installation for each product are listed in the following table.

Product	Delivery and Installation Method
Windows 8.1	<ul style="list-style-type: none"> • DVD • Pre-installed on the computer by OEM • Download that updates Windows 8
Windows Server 2012 R2	<ul style="list-style-type: none"> • DVD • Pre-installed on the computer by OEM • Download that updates Windows Server 2012
Windows Storage Server 2012 R2	<ul style="list-style-type: none"> • Pre-installed by the OEM (Third party)
Surface Pro 2, Surface Pro, Surface 2, Surface	<ul style="list-style-type: none"> • Pre-installed by the OEM (Microsoft)
Windows RT 8.1	<ul style="list-style-type: none"> • Pre-installed on the device by OEM • Download that updates Windows RT
Windows Phone 8.1	<ul style="list-style-type: none"> • Pre-installed on the device by OEM or mobile network operator • Download that updates Windows 8
Windows Embedded 8.1 Industry Enterprise	<ul style="list-style-type: none"> • Pre-installed by the OEM (Third party)
StorSimple 8000 Series	<ul style="list-style-type: none"> • Pre-installed by the OEM (Third party)

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: <http://www.microsoft.com/en-us/howtotell/default.aspx>

The installed version of Windows 8.1 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:

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	
3DES	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.

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:

<http://windows.microsoft.com>

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

<http://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 8.1 OEs must be verified to match the version that was validated using one of the following methods:

1. The ver command
 - a. From Start, open the Search charm.
 - b. In the search field type "cmd" and press the Enter key.
 - c. The command window will open with a "C:\>" prompt.
 - d. At the prompt, type "ver" and press the Enter key.
 - e. You should see the answer "Microsoft Windows [Version 6.3.9600]".
2. The systeminfo command
 - a. From Start, open the Search charm.
 - b. In the search field type "cmd" and press the Enter key.
 - c. The command window will open with a "C:\>" prompt.
 - d. At the prompt, type "systeminfo" and press the Enter key.
 - e. Wait for the information to be loaded by the tool.
 - f. Near the top of the output, you should see:

OS Name:	Microsoft Windows 8.1 Enterprise
OS Version:	6.3.9600 N/A Build 9600
OS Manufacturer:	Microsoft Corporation

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: x.x.xxxx.xxxxx.
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.